

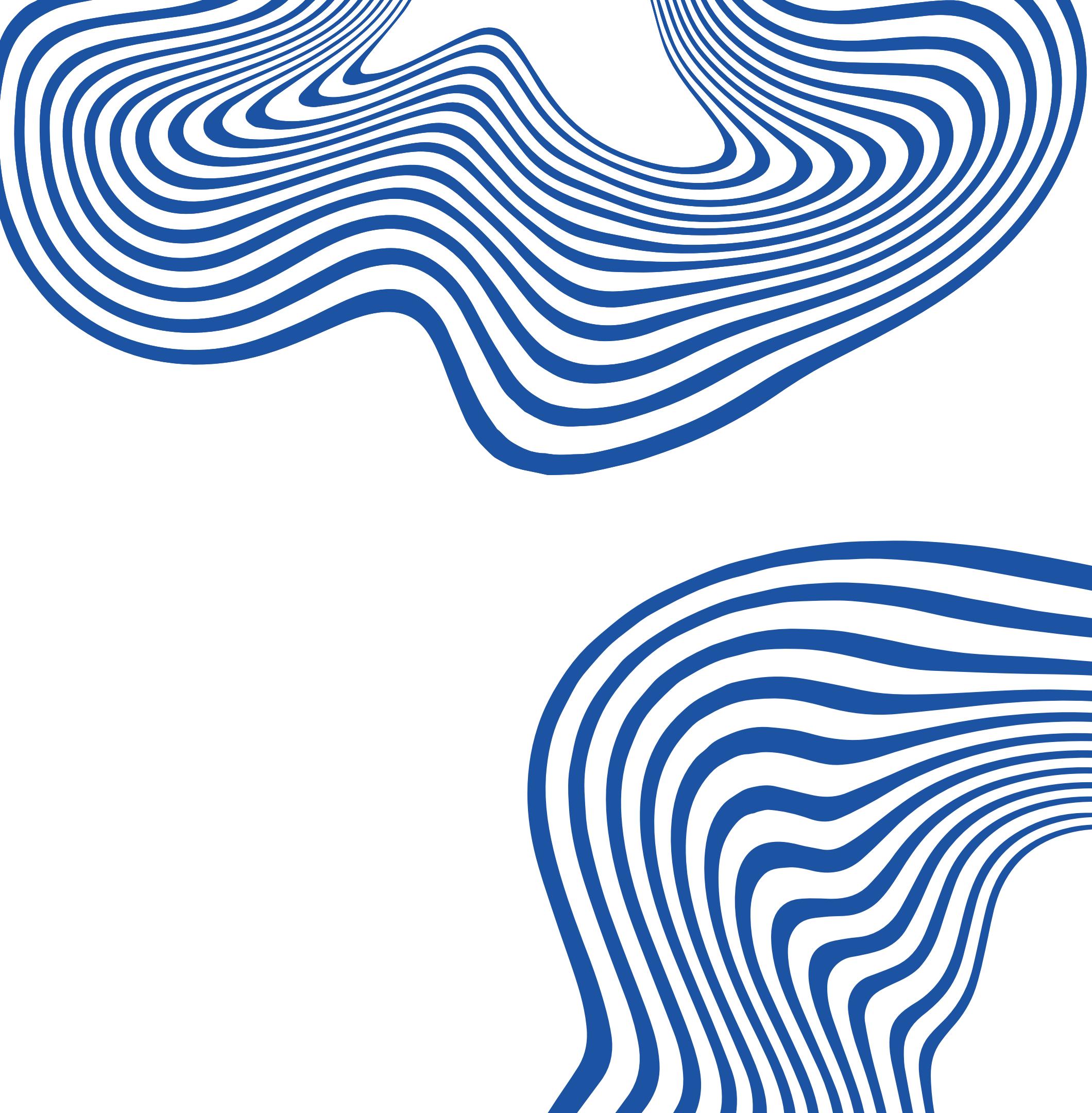


Multi-Layer Perceptron

Fontana Giorgia - 43206A

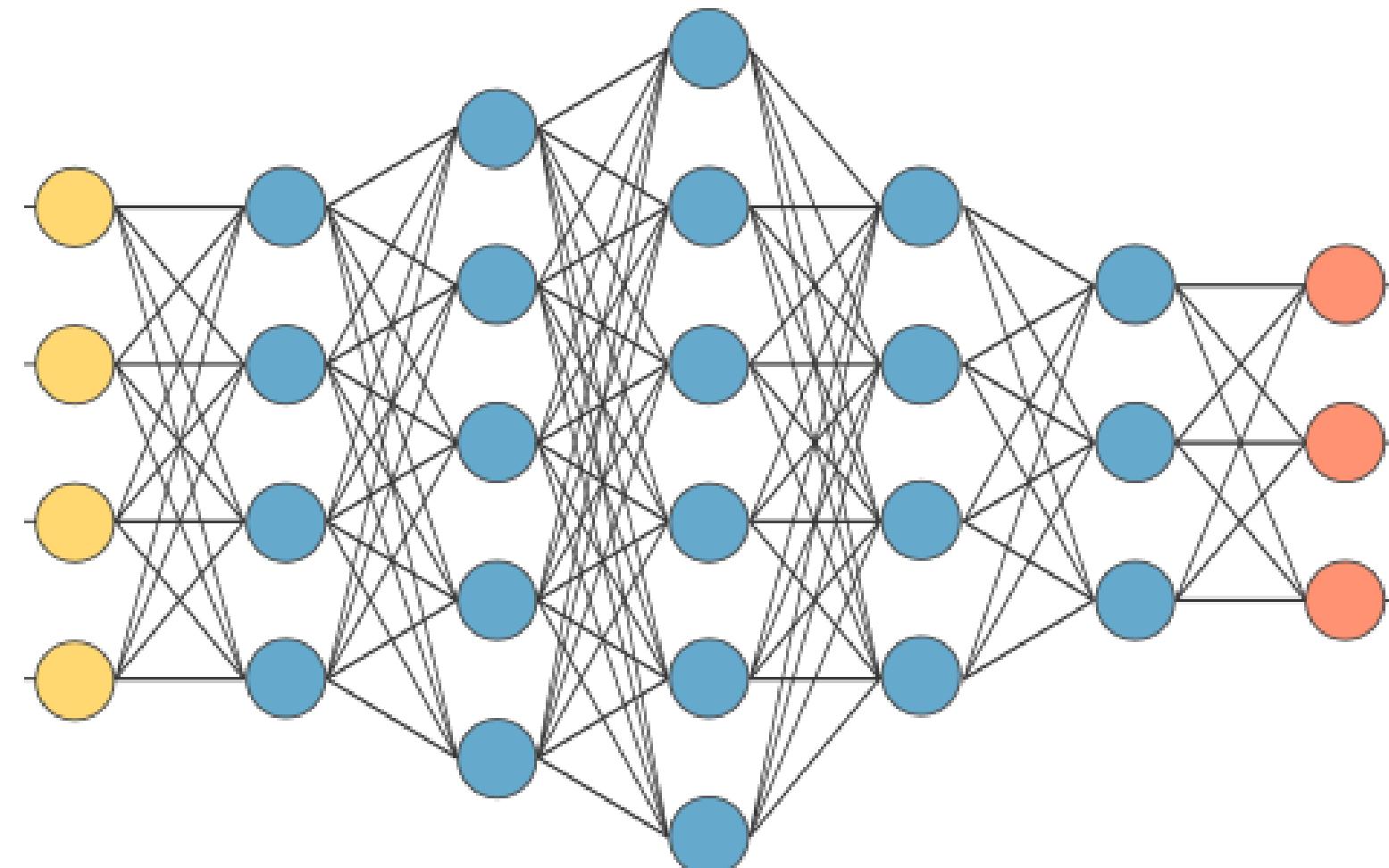
Verga Wiktor - 41644A

Zanchetta Elena - 82066A



Artificial Neural Network (NN)

Una NN è un **modello matematico** ispirato al funzionamento del **cervello umano**. È composta da **neuroni artificiali interconnessi** che trasformano degli input in output attraverso pesi e funzioni di attivazione.

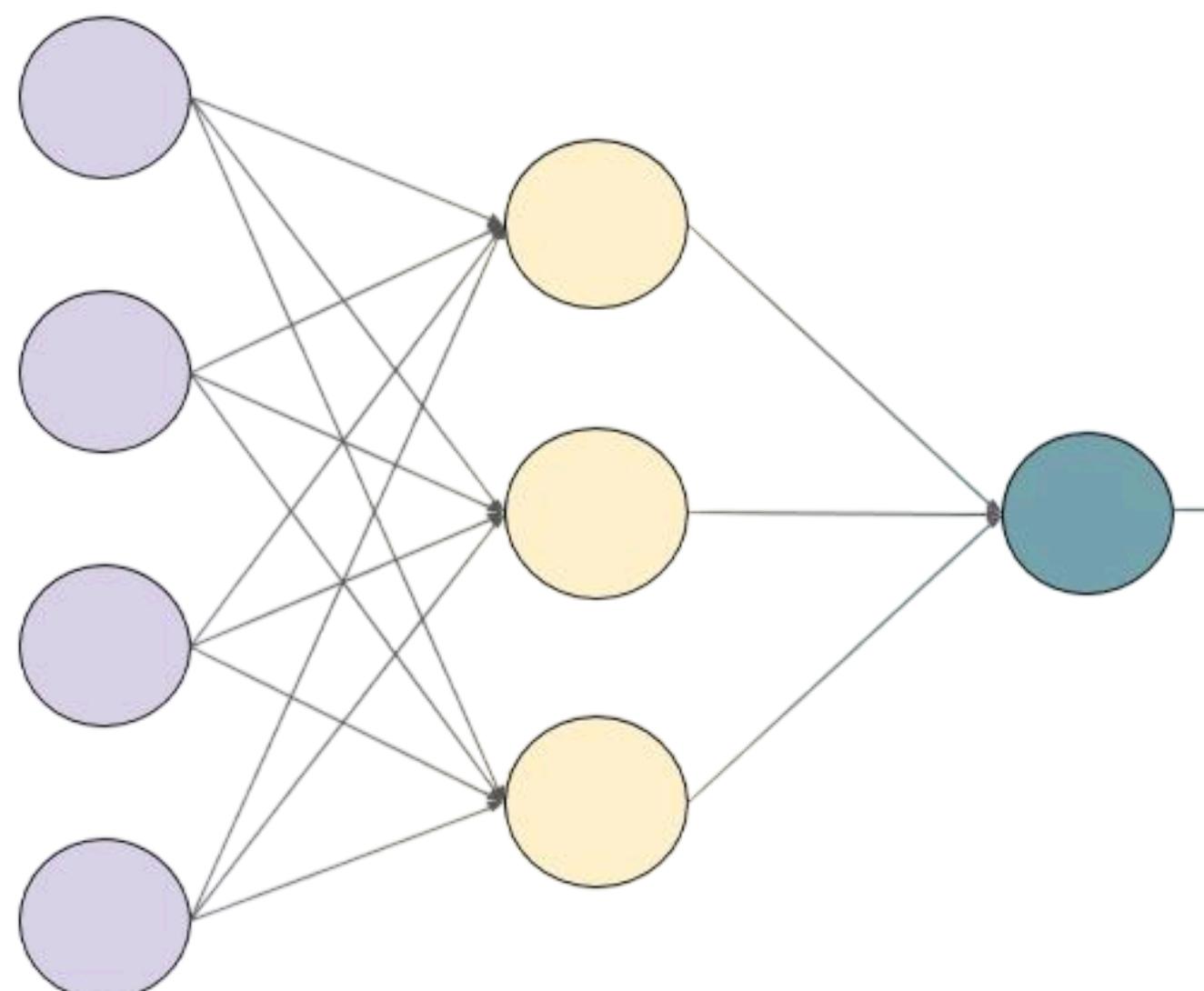


Classificazione Neural Network



Feed-Forward Neural Network (FNN)

Le **FNN** sono reti in cui i dati scorrono solo in avanti:
input → hidden layer → output
Sono la **base teorica** di quasi tutte le architetture moderne.
Le **MLP (Multi-Layer Perceptron)** sono una tipologia di FNN



Convolutional Neural Networks (CNN)

Le **CNN** sono un tipo di FNN, progettate specificamente per le **immagini**. Usano la **convoluzione**, un'operazione che usa un **filtro** per rilevare pattern locali come bordi, forme e texture. Sono fondamentali per computer vision e riconoscimento immagini.

Recurrent Neural Network (RNN)

Le **RNN** derivano dalle FNN ma aggiungono **connessioni ricorrenti**. Hanno una **memoria interna** che permette di trattare sequenze. Variante: **LSTM (Long Short-Term Memory)**, introduce i gate, che controllano il flusso dell'informazione.



Modelli Generativi

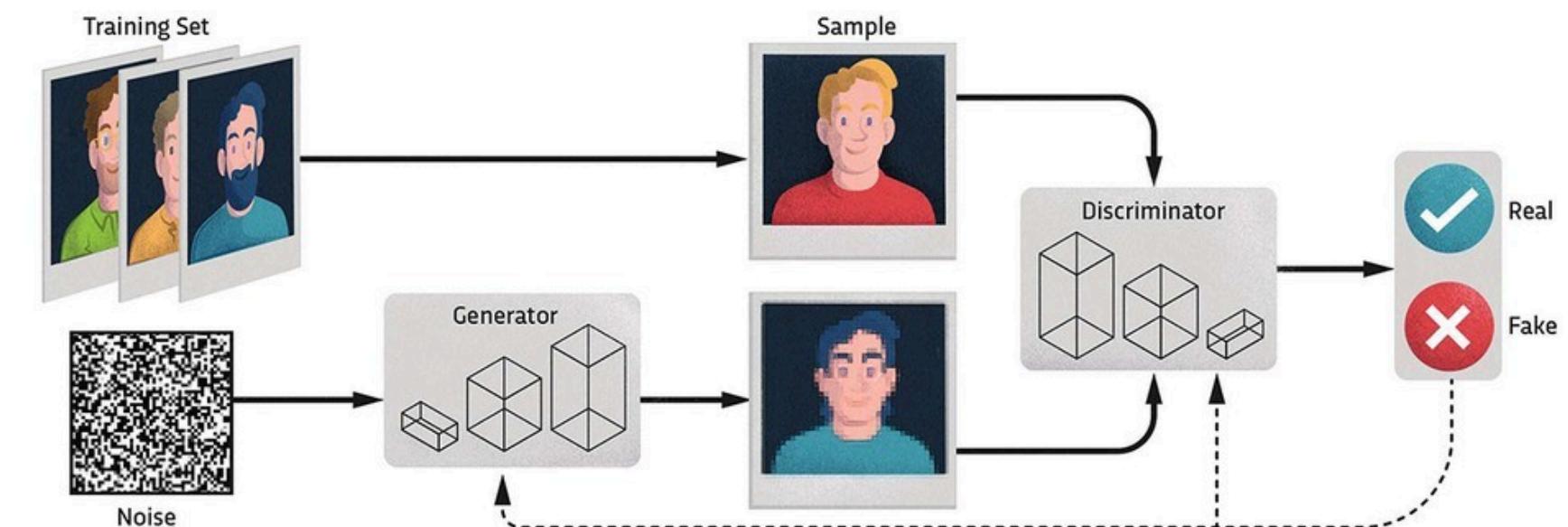
Generative Adversarial Networks (GAN)

Le **GAN** non derivano dalle FNN.

Sono un **framework** composto da due reti che competono:

- **Generatore**: crea dati realistici
- **Discriminatore**: distingue dati reali da falsi

Sono utilizzate per generare immagini e deepfake.

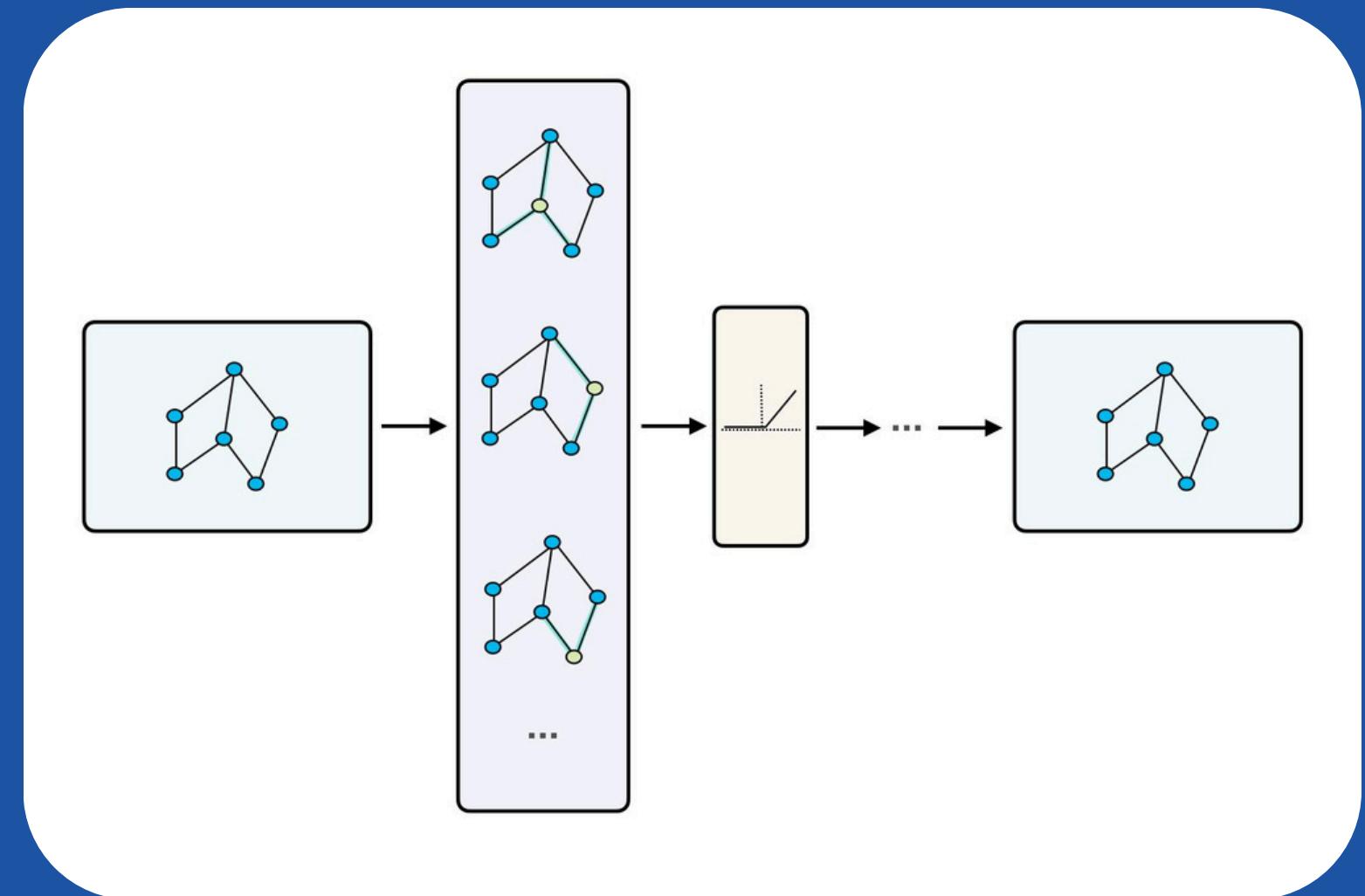


Graph Neural Networks (GNN)

Le **GNN** sono progettate per lavorare su **grafi**, cioè dati strutturati con **nodi** e **connessioni**.

Ogni nodo aggiorna le proprie informazioni tramite **propagazione** dai vicini.

Applicazioni: social network, molecole e traffico

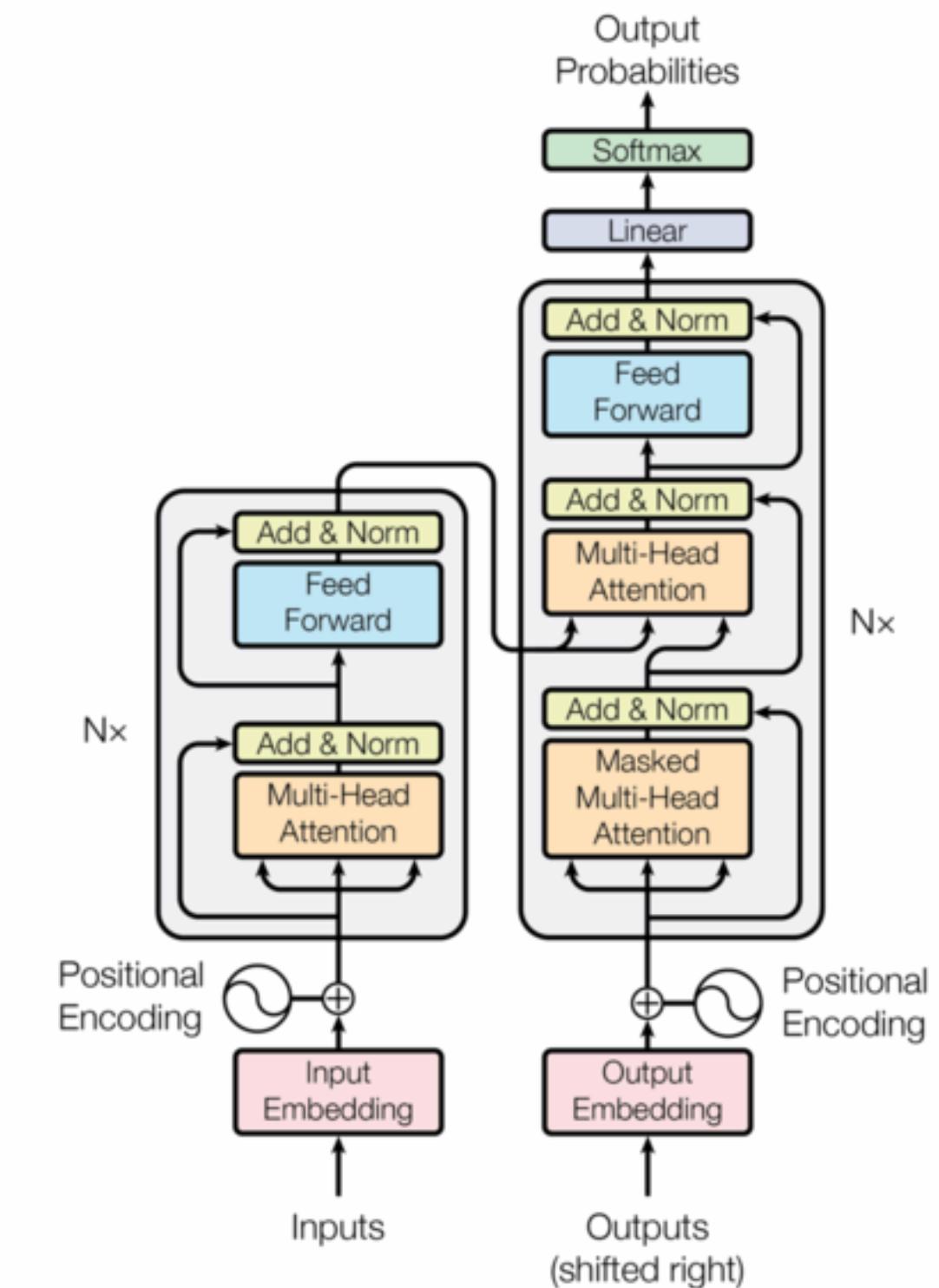


Transformer

I **Transformer** si basano sul meccanismo di **self-attention**, che permette a ogni elemento della sequenza di “vedere” tutti gli altri.

Componenti principali:

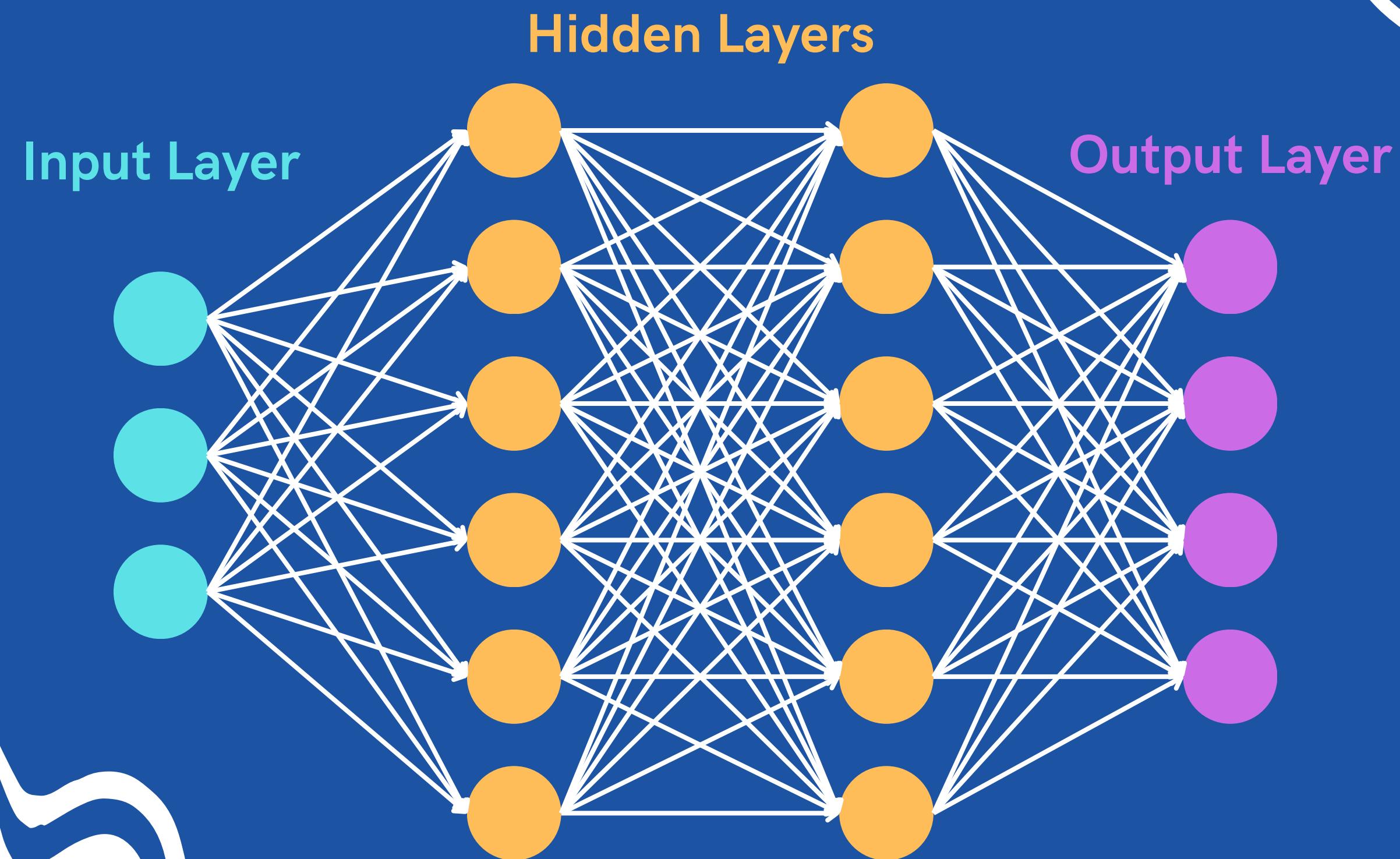
- **Multi-Head Attention**: parallelizza il self-attention
- **Residual Connections + Layer Norm**: stabilizzano il calcolo e permettono reti molto profonde
- **Positional Encoding**: aggiunge informazioni sull’ordine degli elementi



Funzionamento di una Neural Network MLP



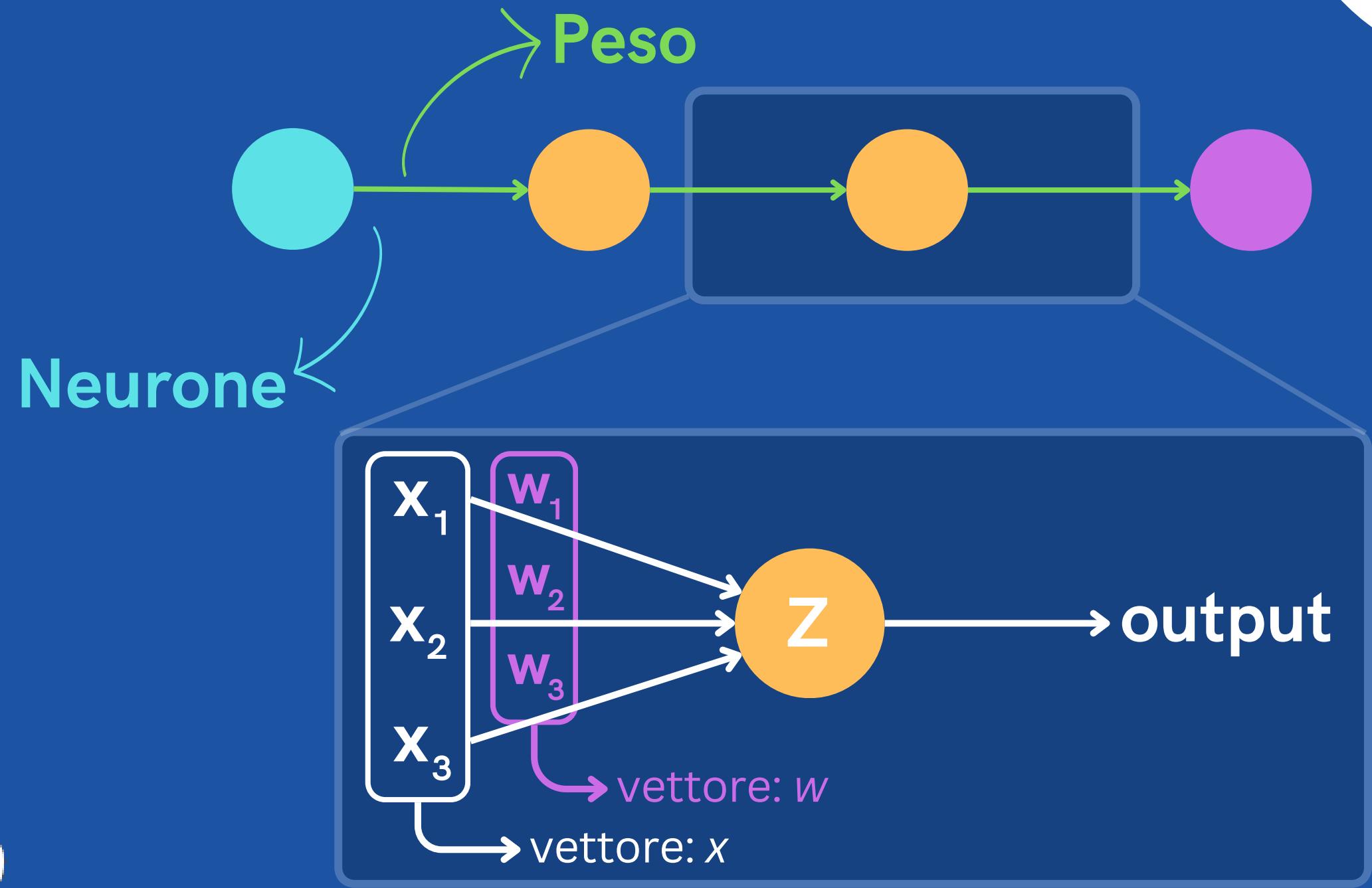
Struttura MLP



Perceptron

- **Input:** numero reale x
- **Peso:** numero reale w
 - Indica l'importanza dell'input
- **Bias:** numero reale b
 - *Soglia di attivazione*
- **Output:** 1 oppure 0

$$z = w \cdot x + b$$



$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

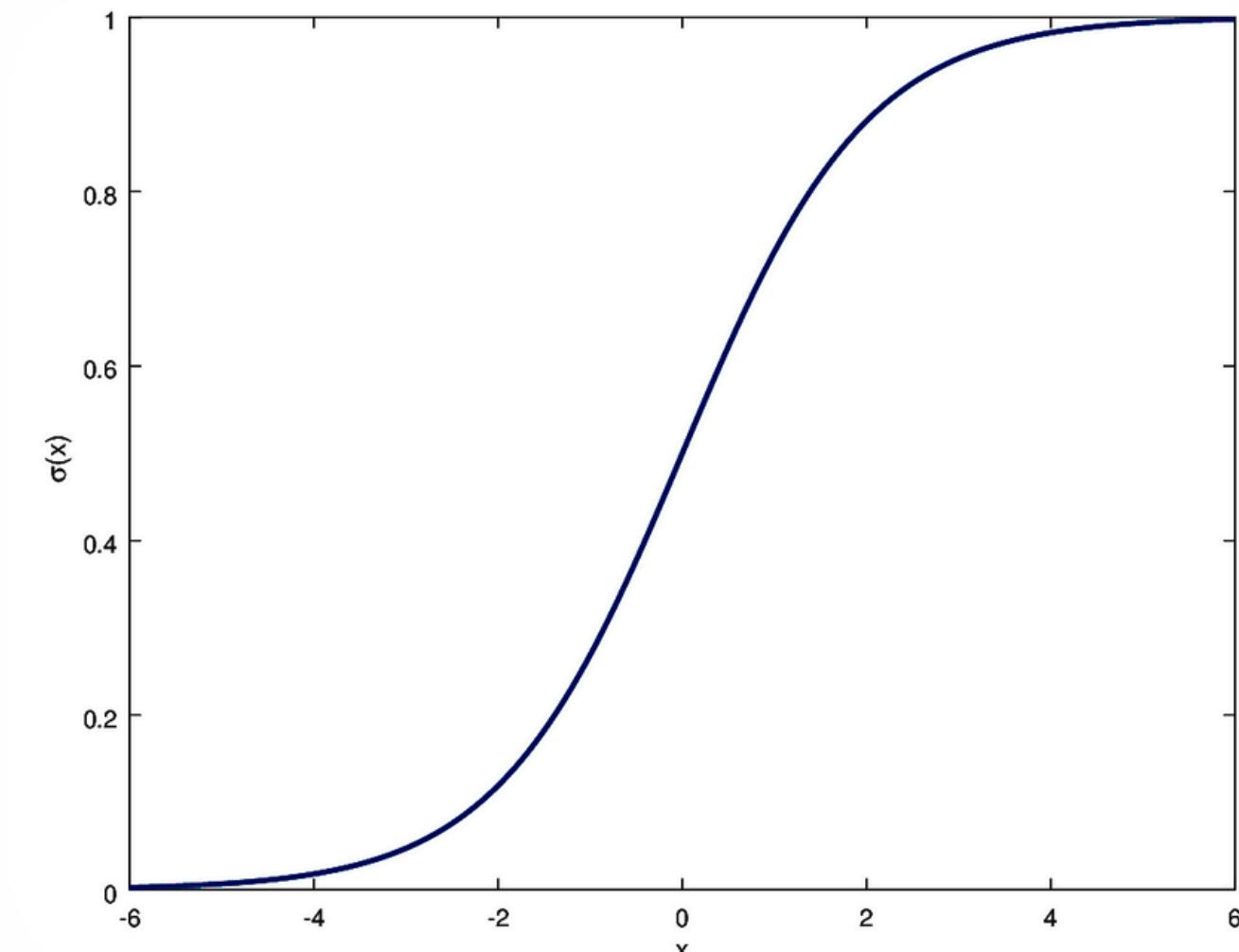
Sigmoid Neuron

Con il Perceptron il risultato è binario: 0 o 1.

Il Sigmoid permette **più controllo** nell'output, dando come risultato un numero reale compreso tra 0 e 1

L'output viene definito utilizzando una **funzione d'attivazione**

Esistono molte altre funzioni d'attivazione



$$\sigma(w \cdot x + b)$$

Training

Come capisce un neurone
l'importanza di un input?

Fasi del Training

Epoca

Dato in Input 1

Forward Pass

Calcolo dell'Errore

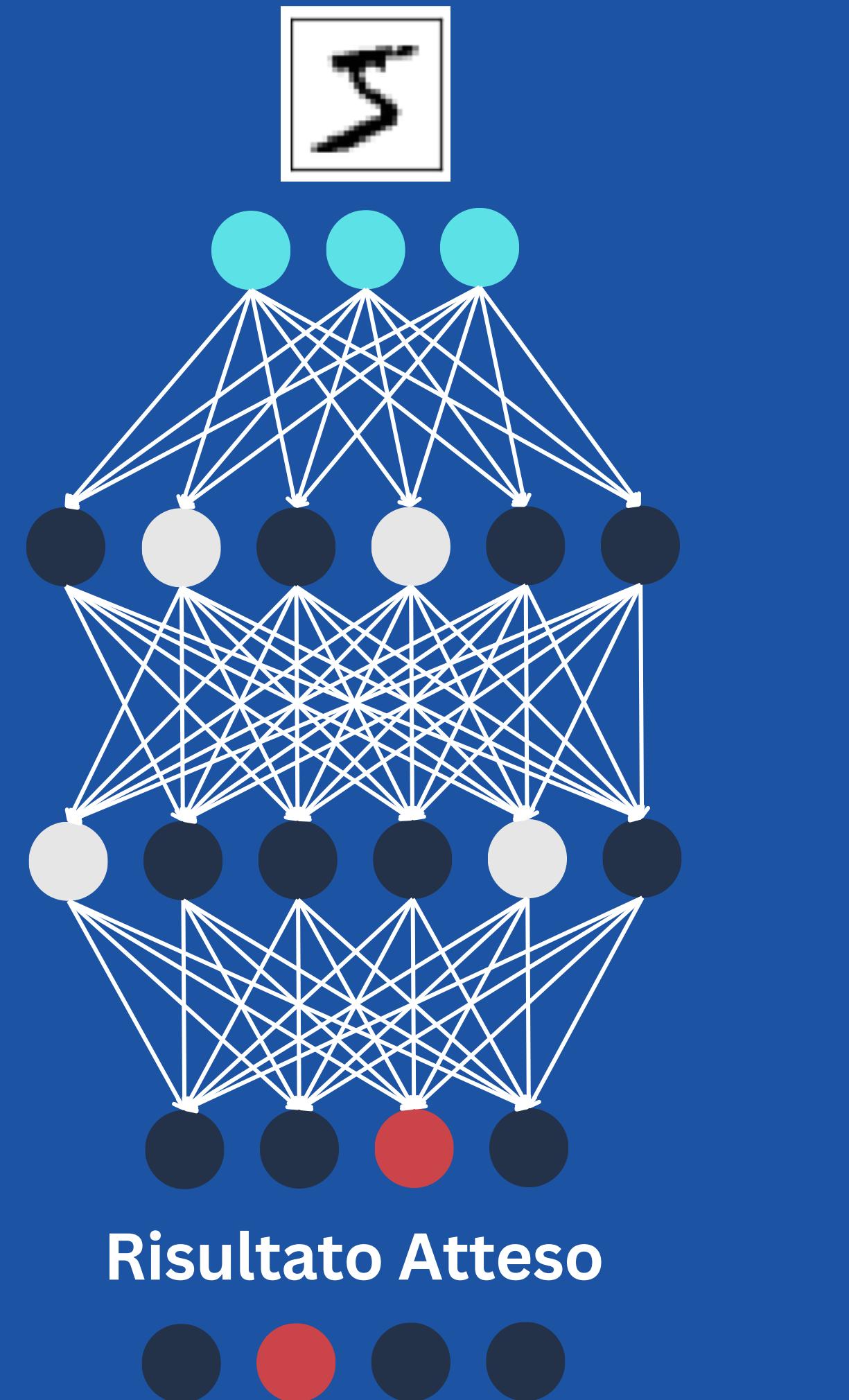
Backpropagation

Gradient Descent

Dato in Input 2

...

Dato in Input n



Forward Pass

Il dato di input viene fatto passare nella NN e viene calcolato il risultato

Calcolo dell'Errore

Il risultato ottenuto viene paragonato a quello atteso e si calcola l'errore utilizzando una

Loss Function:

- Mean Square Error
- Cross Entropy

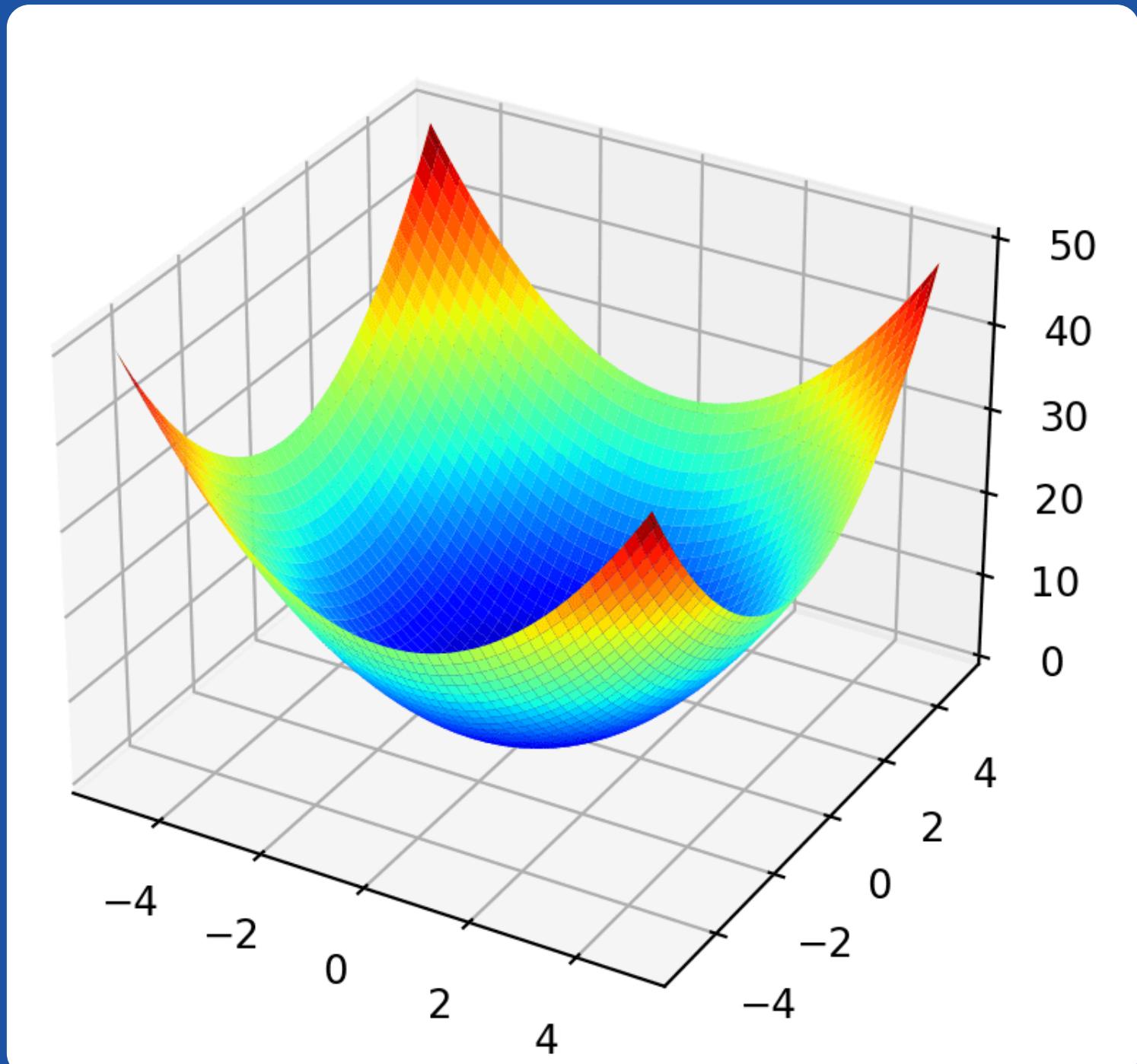


Come imparano le NN

Una volta calcolato l'errore sappiamo di quanto la rete ha sbagliato

Cercando di **minimizzare** il più possibile questo **errore** siamo in grado di **modellare il comportamento** della rete insegnandole come raggiungere i **risultati corretti**

Il problema si riduce ad un problema di minimizzazione della **Loss Function**

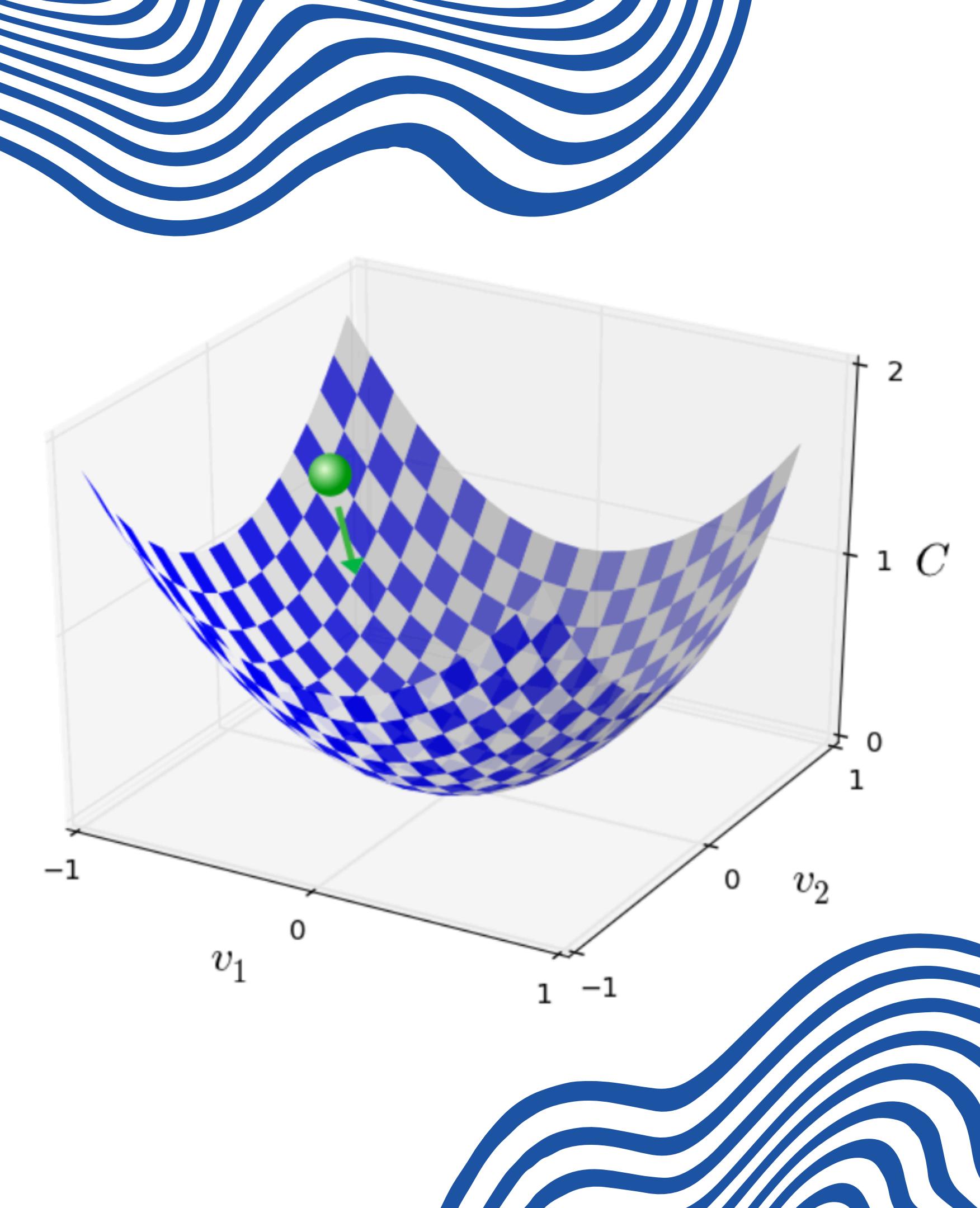


Gradient Descent

Permette di ottimizzare il calcolo del **minimo** andando a ridurre la quantità di **derivate** da calcolare per avvicinarci al punto minimo

Permette di calcolare il punto minimo anche nel caso di una funzione con **molti parametri** (caso tipico in una NN)

Si ha il minimo di una funzione quando la **derivata** è uguale a 0



Gradient Descent

Il *Gradiente* è il **vettore delle derivate** parziali di una funzione rispetto ai suoi parametri

La *Discesa* consiste nel calcolare il gradiente in punti diversi della funzione secondo un certo **passo**

Il *Passo* è determinato dal valore del gradiente calcolato precedentemente e il **Learning Rate**

Il Gradiente

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T.$$

Calcolo del nuovo valore

$$v \rightarrow v' = v - \eta \nabla C.$$

Algoritmo del Gradient Descent

01

Calcolare il
gradiente
della Loss
Function

02

Scegliere dei
valori
casuali per
parametri

03

Calcolare il
gradiente per
i vari
parametri

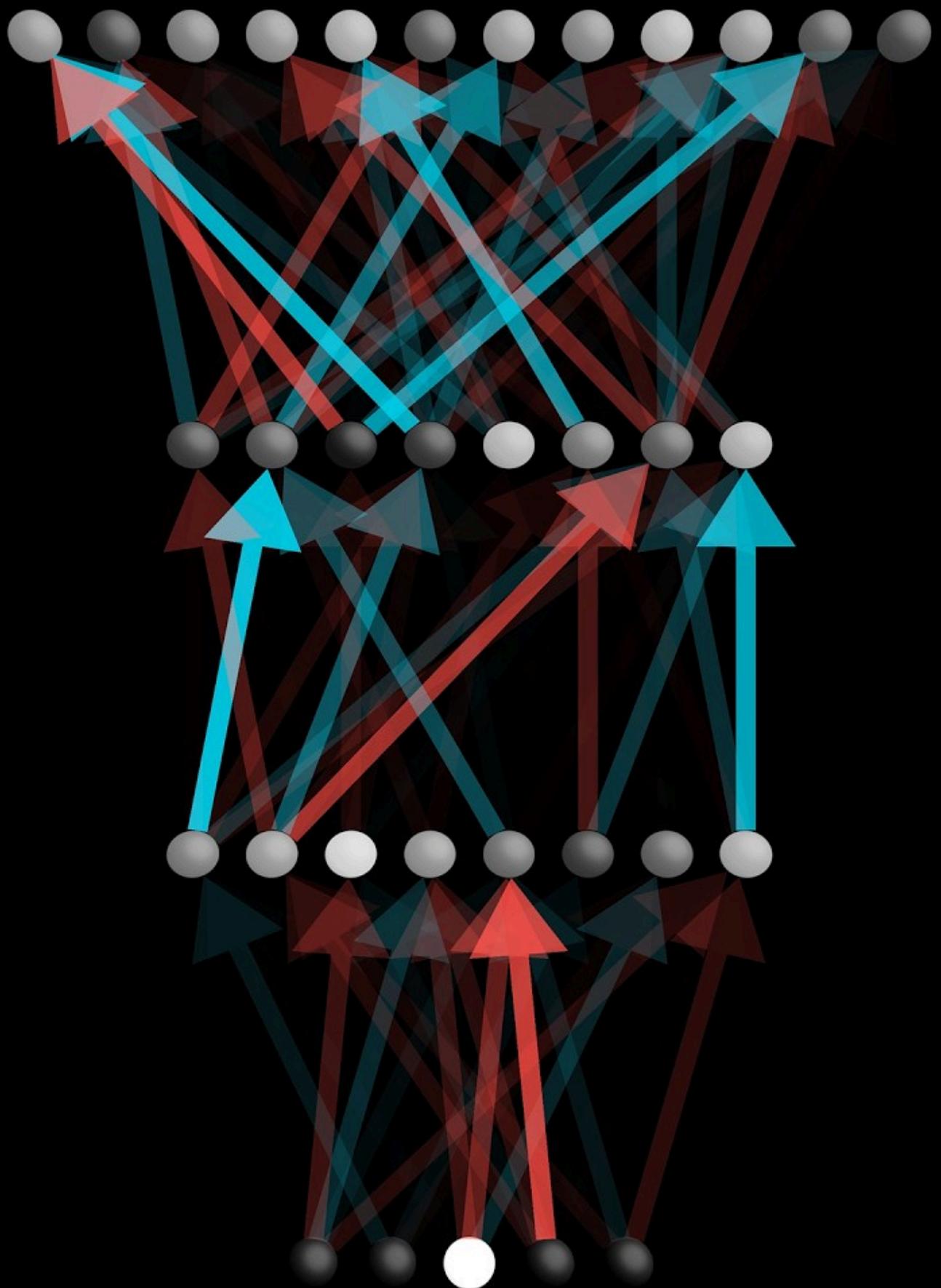
04

Calcolare il
passo per il
prossimo
valore

05

Calcolare i
nuovi valori dei
parametri
usando il passo





Backpropagation

Si occupa di capire quanto ogni peso e bias abbia **contribuito all'errore finale** della rete neurale.

Calcola i gradienti della Loss Function **rispetto** a ogni peso e bias della rete.

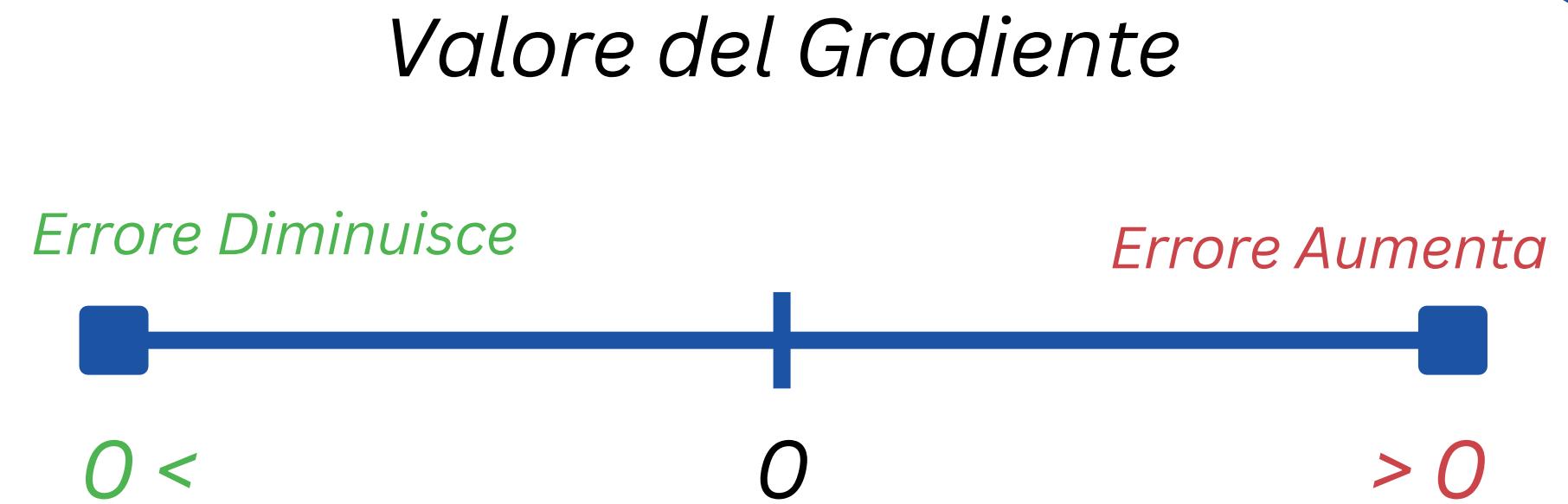
Questi gradienti indicano quanto una piccola variazione di quel parametro influirebbe sull'errore finale



Chain Rule & Cambio dei Pesi

I gradienti dei singoli parametri
possono essere calcolati
scomponendo le derivate
dell'errore in una serie di **derivate
parziali** *collegate* tra loro

Così il GD sarà in grado di **capire**
quanto modificare il peso o il
bias a seconda della **direzione**
che prende il gradiente.



*peso è da
aumentare*

*peso non
influisce*

*peso è da
diminuire*

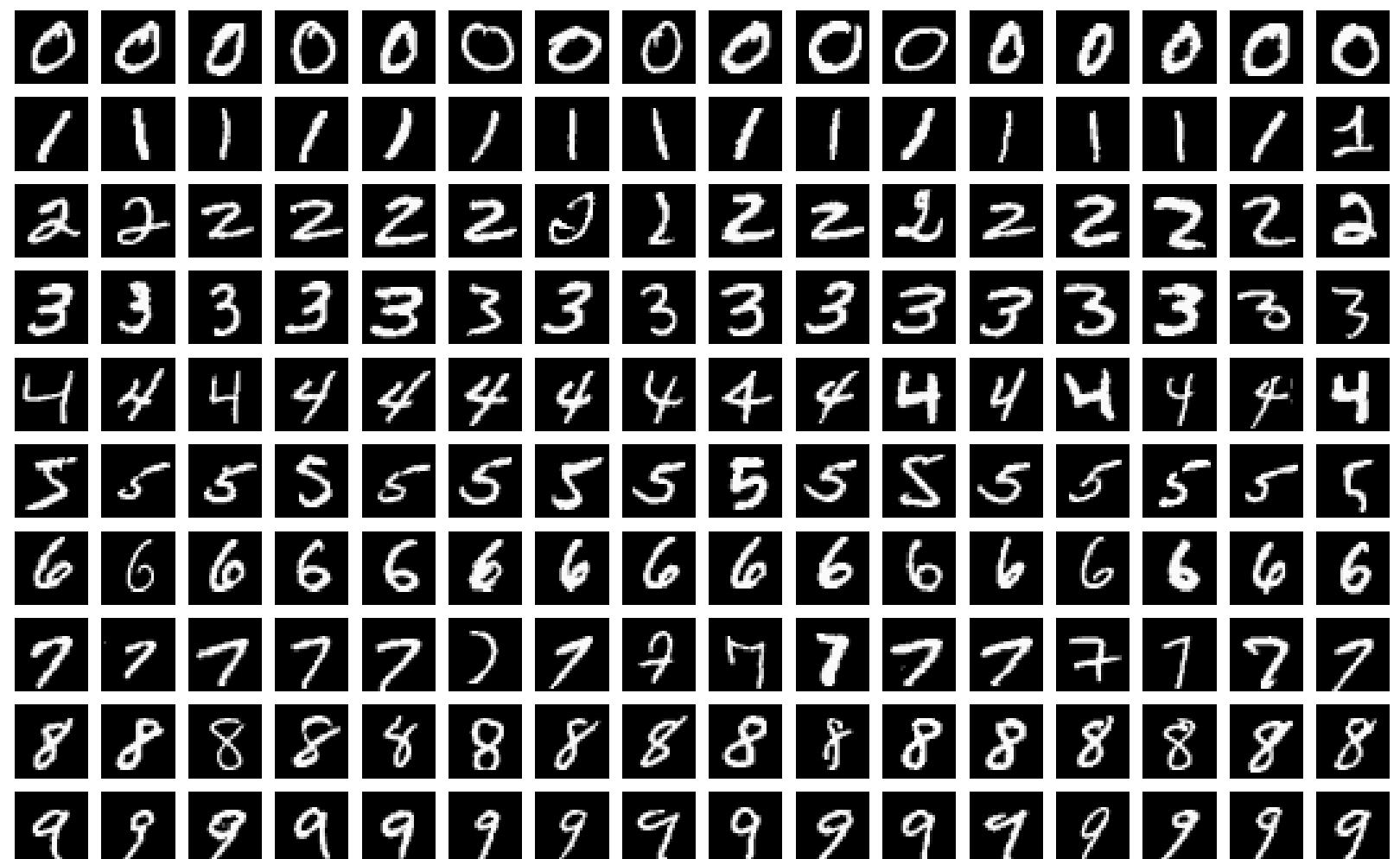
Multi-Layer Perceptron del dataset MNIST



```
8 # --- 1. CARICAMENTO E PREPARAZIONE DATI ---
9 print("Caricamento dataset MNIST...")
10 # Il dataset è composto da 60.000 immagini di training e 10.000 di test e tutte le immag
11 # in scala di grigi
12 (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

Dataset MNIST

Il **dataset MNIST** serve per riconoscere cifre scritte a mano. Contiene immagini in scala di grigi a bassa risoluzione, ognuna rappresentante una **cifra da 0 a 9**.



```
14 # Normalizzazione: dividiamo valori dei pixel dividendo per 255 → ottenendo valori tra 0  
15 # Questo aiuta la rete neurale a convergere (imparare) più velocemente.  
16 x_train = x_train / 255.0  
17 x_test = x_test / 255.0
```

Normalizzazione dei Pixel

I pixel delle immagini vanno da 0 a 255, dividendo i valori dei pixel per 255 otteniamo **valori compresi tra 0 e 1**.

Le reti neurali **imparano molto meglio** quando i numeri in **input sono piccoli e ben scalati**. Rende l'addestramento più veloce e più stabile.



Costruzione del modello

```
21 # --- 2. COSTRUZIONE DEL MODELLO (LA RETE NEURALE) ---
22 # Usiamo un modello "Sequenziale": i dati passano da un layer all'altro.
23 model = keras.Sequential([
24     # Layer 1: Appiattisce l'immagine 28x28 in un vettore di 784 numeri
25     keras.layers.Flatten(input_shape=(28, 28)),
26
27     # Layer 2 (Hidden): 128 neuroni. 'relu' è la funzione di attivazione ( $f(x)=\max(0,x)$ )
28     # Serve a far imparare pattern complessi.
29     # (simile a come un neurone biologico decide se "sparare" o no).
30     keras.layers.Dense(128, activation='relu'),
31
32     # Layer 3 (Output): 10 neuroni (uno per ogni cifra da 0 a 9).
33     # 'softmax' trasforma i risultati in percentuali di probabilità.
34     keras.layers.Dense(10, activation='softmax')
35 ])
```

Costruzione del modello

```
24 |     # Layer 1: Appiattisce l'immagine 28x28 in un vettore di 784 numeri  
25 |     keras.layers.Flatten(input_shape=(28, 28)),
```

Il **primo layer di input** della rete prende ogni immagine e la “appiattisce”, **trasformandola in un vettore** di 784 numeri.

Questo passaggio è necessario perché i **livelli successivi** della rete lavorano con vettori, non con immagini.

Costruzione del modello

```
26  
27     # Layer 2 (Hidden): 128 neuroni. 'relu' è la funzione di attivazione (f(x)=max(0,x))  
28     # Serve a far imparare pattern complessi.  
29     # (simile a come un neurone biologico decide se "sparare" o no).  
30     keras.layers.Dense(128, activation='relu'),  
31
```

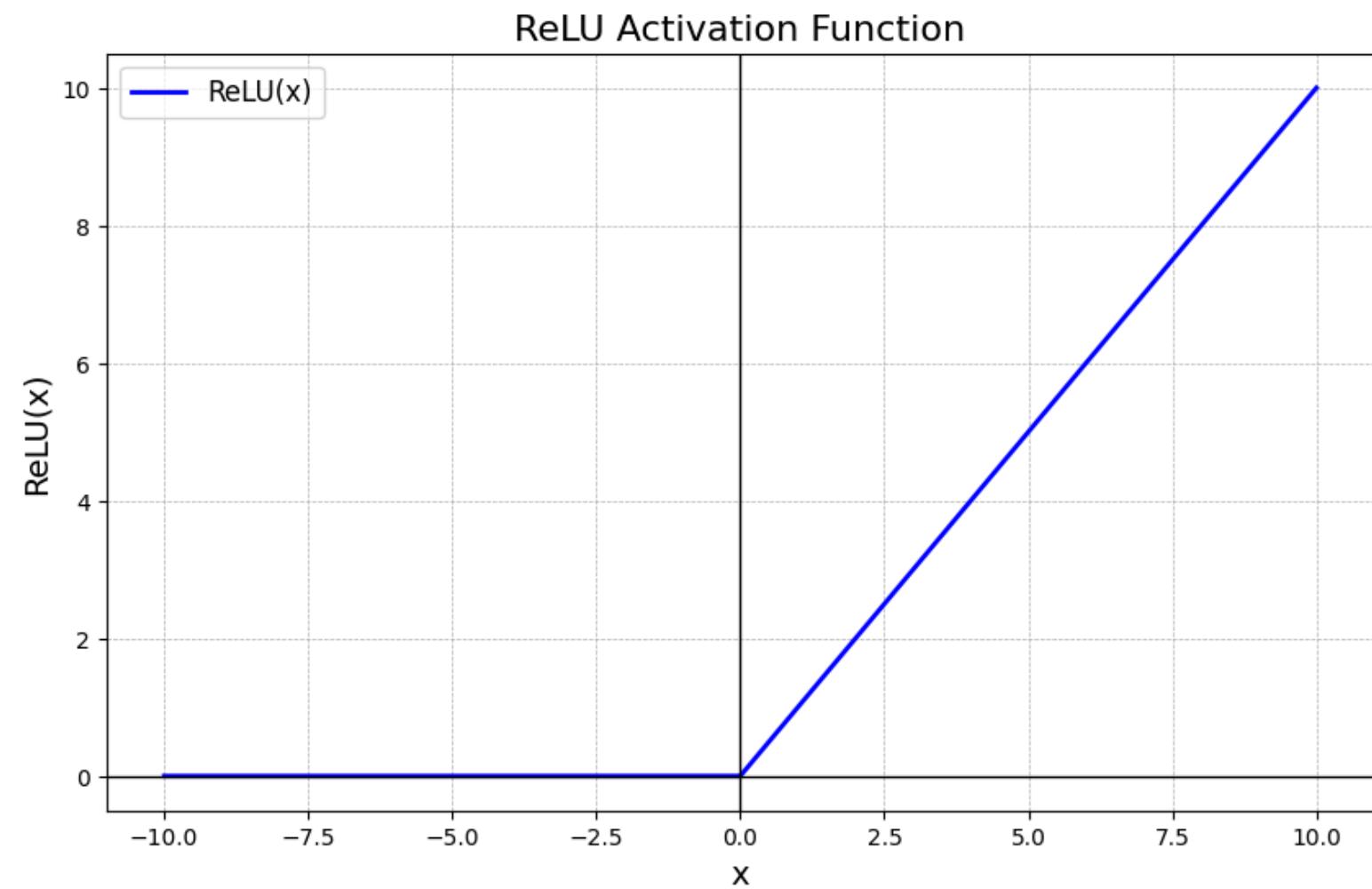
Viene aggiunto il secondo layer della rete neurale: l'**hidden layer**.

Questo strato contiene **128 neuroni**, ognuno dei quali riceve il vettore di 784 numeri generato dallo strato precedente.

La **funzione di attivazione** usata è la ReLU.

Rectified Linear Unit (ReLU)

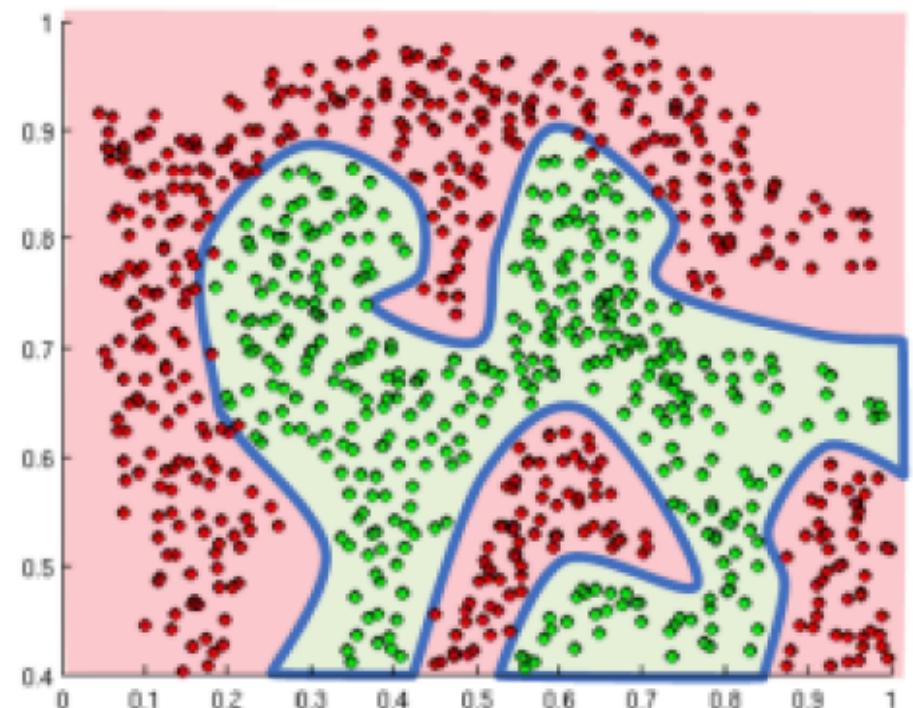
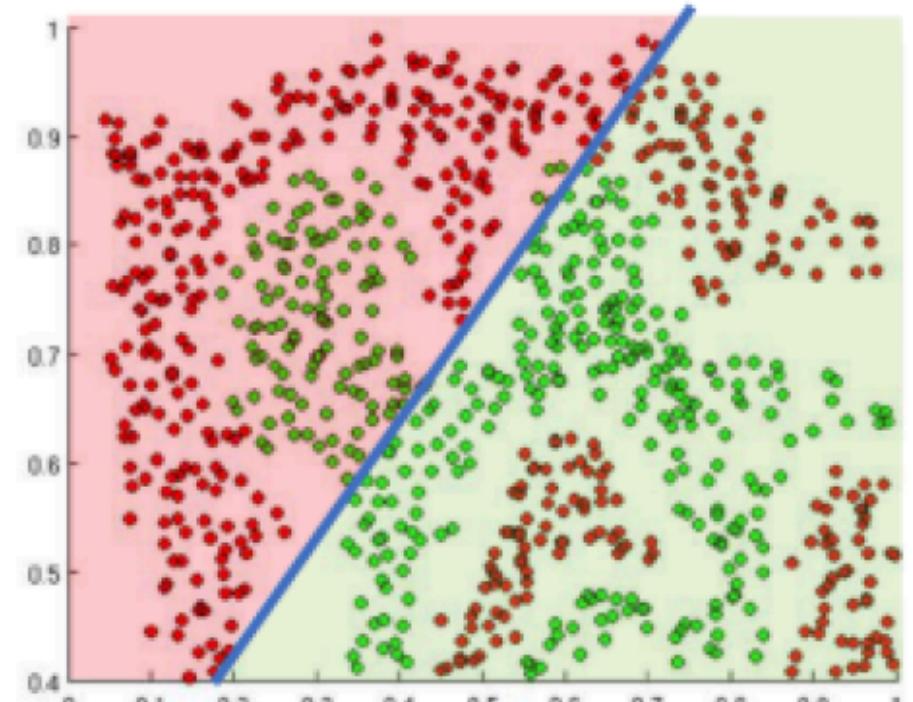
$$ReLU(x) = \max(0, x)$$



Funzione di
attivazione
lineare



ReLU



Costruzione del modello

```
32     # Layer 3 (Output): 10 neuroni (uno per ogni cifra da 0 a 9).  
33     # 'softmax' trasforma i risultati in percentuali di probabilità.  
34     keras.layers.Dense(10, activation='softmax')  
35 ])
```

Viene aggiunto l'ultimo strato della rete, il **layer di output**, composto da **10 neuroni**.

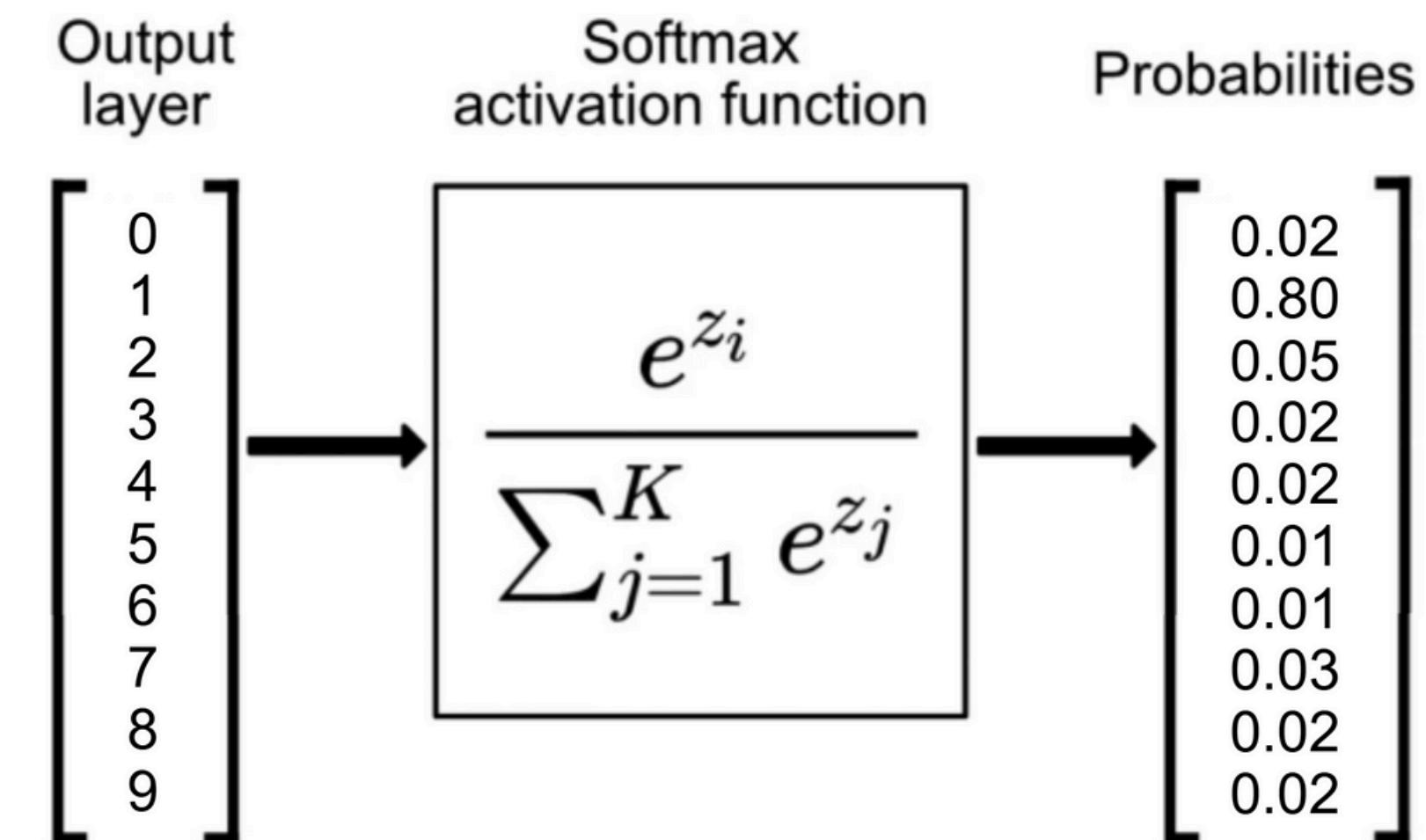
Poiché il dataset MNIST contiene cifre da 0 a 9, servono **10 categorie** possibili.

La funzione utilizzata è **softmax**.

Softmax

Softmax assegna a ciascuna classe una probabilità.

Il valore di uscita di un neurone dipende anche da tutti gli altri neuroni dello stesso strato.



Compilazione del modello

```
37 # Compilazione del modello  
38 #Definisci come la rete impara:  
39 #Adam: ottimizzatore intelligente che aggiorna i pesi  
40 #cross-entropy: misura l'errore di classificazione  
41 #accuracy: metrica per valutare quanto indovina  
42 model.compile(optimizer='adam',  
43                 loss='sparse_categorical_crossentropy',  
44                 metrics=['accuracy'])
```

Inizia la **compilazione del modello**.

In questa parte del codice il modello non viene ancora addestrato, ma gli viene spiegato cosa deve imparare.

```
42     model.compile(optimizer='adam',  
43                     loss='sparse_categorical_crossentropy',  
44                     metrics=['accuracy'])
```

Adam: adaptive moment estimation

L'**ottimizzatore** è l'algoritmo che **aggiorna iterativamente i pesi** della rete neurale in base all'errore calcolato, guidando il modello verso la **minimizzazione della funzione di loss**.

A differenza dei metodi con Learning Rate fisso, **Adam** utilizza un approccio **adattivo**, assegnando un **Learning Rate** diverso a ciascun parametro in base al comportamento dei gradienti.

```
42     model.compile(optimizer='adam',  
43                     loss='sparse_categorical_crossentropy',  
44                     metrics=['accuracy'])
```

Loss function

“Sparse categorical cross entropy” è la nostra funzione di errore, è utilizzata quando le **classi sono mutuamente esclusive e sono numeri interi**, proprio come nel dataset MNIST.

```
42 model.compile(optimizer='adam',  
43                 loss='sparse_categorical_crossentropy',  
44                 metrics=['accuracy'])
```

Metrics

Metrics è ciò che vogliamo monitorare per capire se il modello sta imparando bene.

In questo caso specifichiamo che vogliamo monitorare **l'accuratezza**. L'accuracy indica quante immagini il modello riconosce correttamente su 100.

Addestramento del modello



```
46 # --- 3. ADDESTRAMENTO ---
47 print("\nInizio addestramento...")
48 # epochs=5 significa che la rete vedrà tutto il dataset per 5 volte.
49 history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

Inizia il training del modello:

La rete neurale, durante ogni epoca, cerca di indovinare che numero c'è nell'**immagine di training** e dopo ogni tentativo confronta la sua risposta con quella reale, calcola l'errore e aggiorna i pesi interni per migliorare.

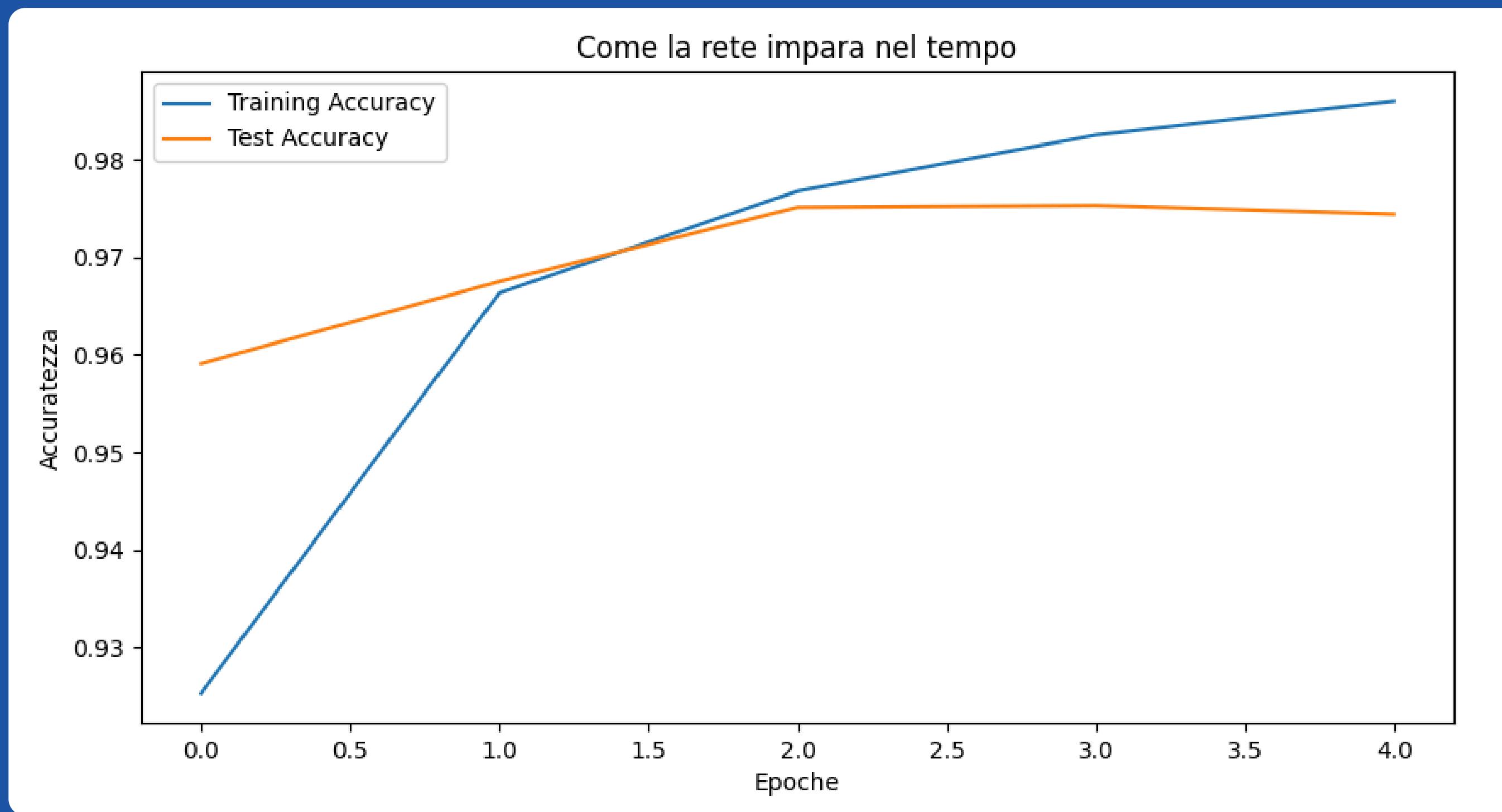
Dopo ogni epoca, il modello viene testato sulle **immagini di test**.

Questo permette di verificare se sta davvero imparando o se sta semplicemente memorizzando gli esempi.

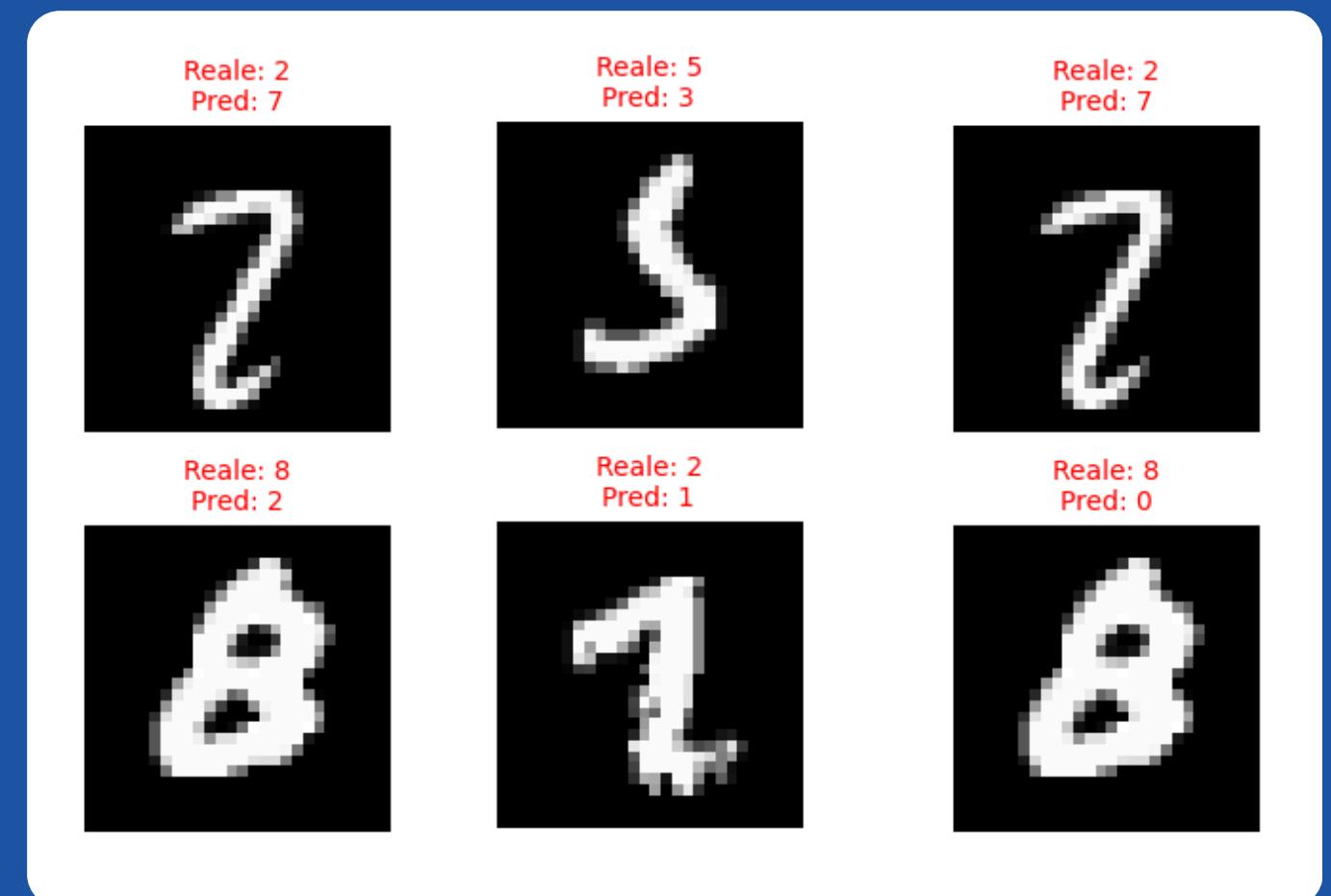
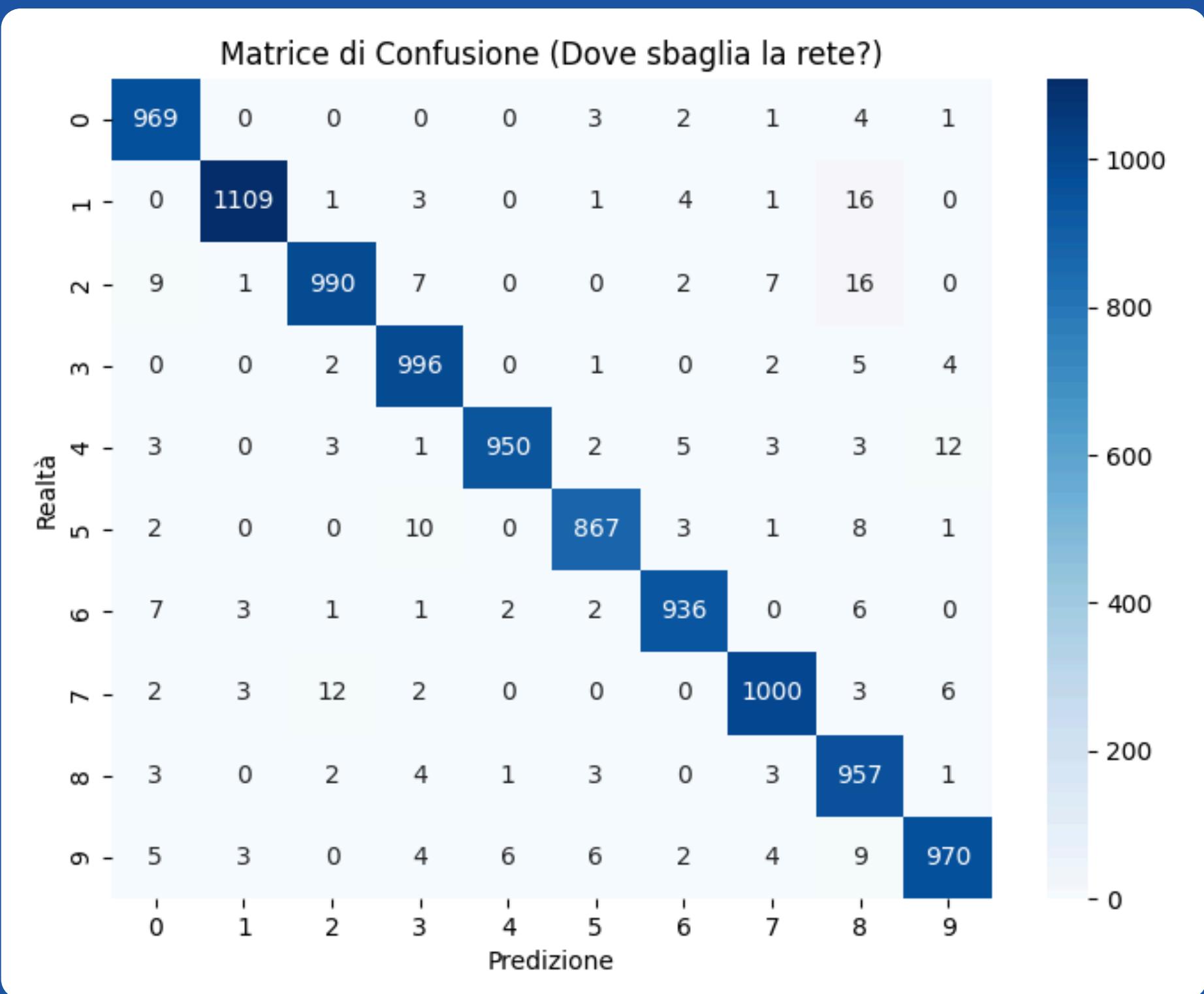


Risultati
Ottenuti

Curva d'Apprendimento



Matrice di Confusione

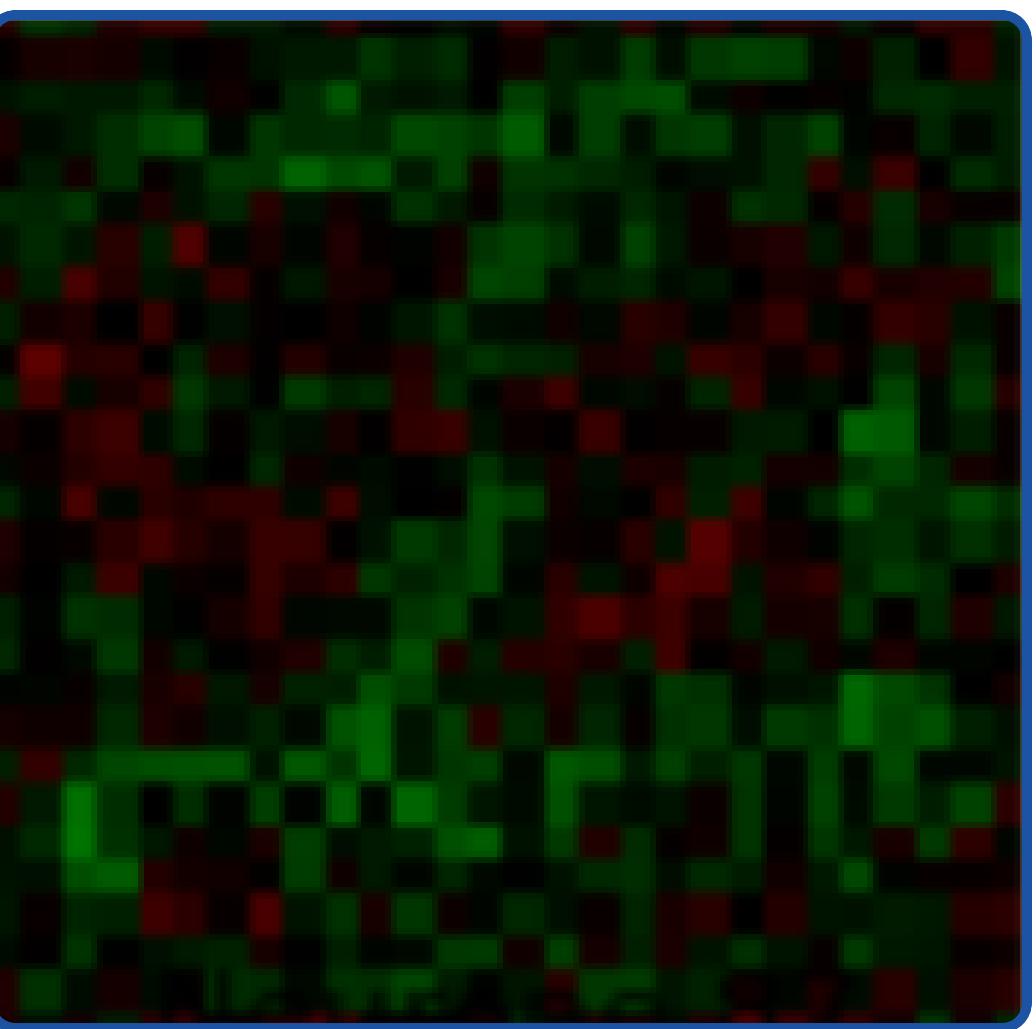


Cosa vedono i Neuroni

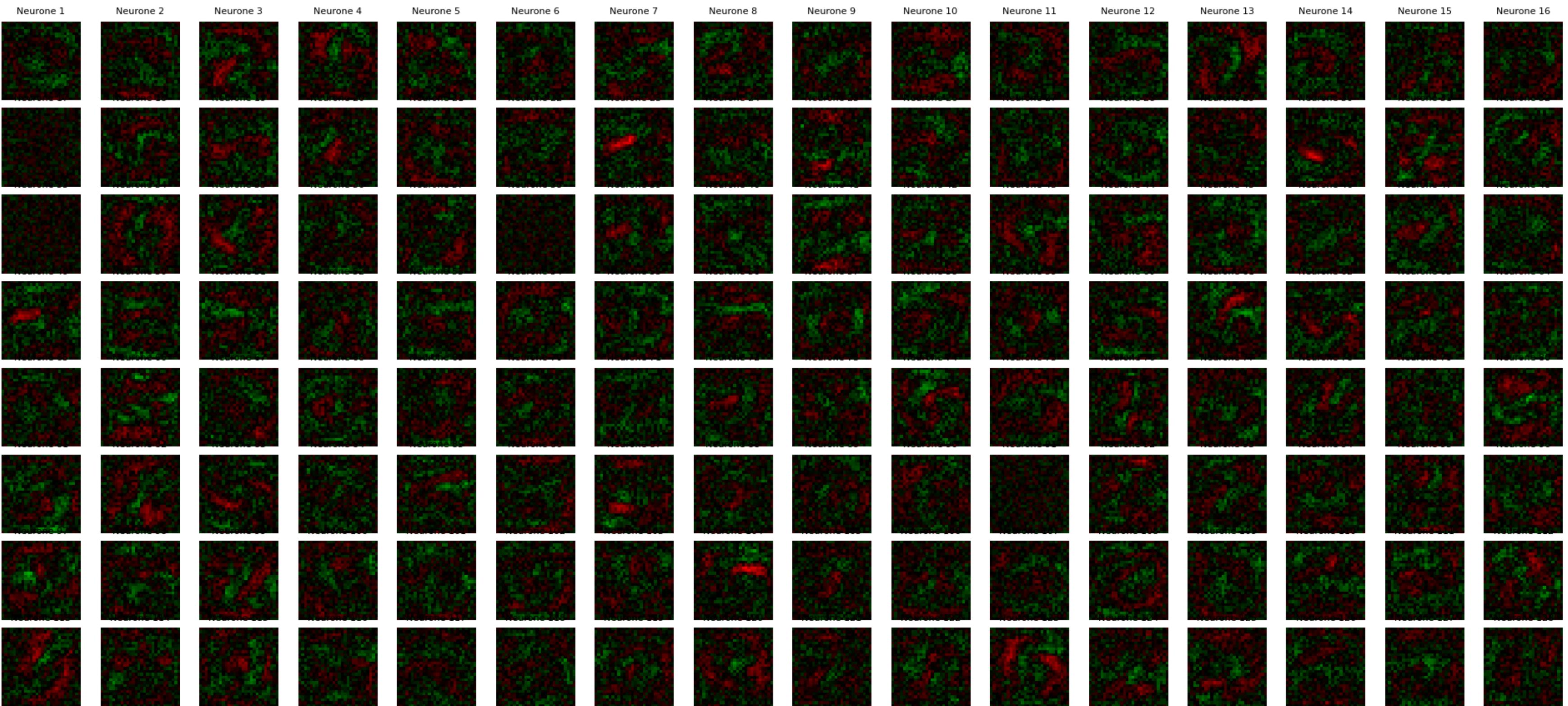


Possiamo osservare i pattern appresi dai neuroni durante il training, visualizzando i pesi (W) che connettono l'input ai 128 neuroni

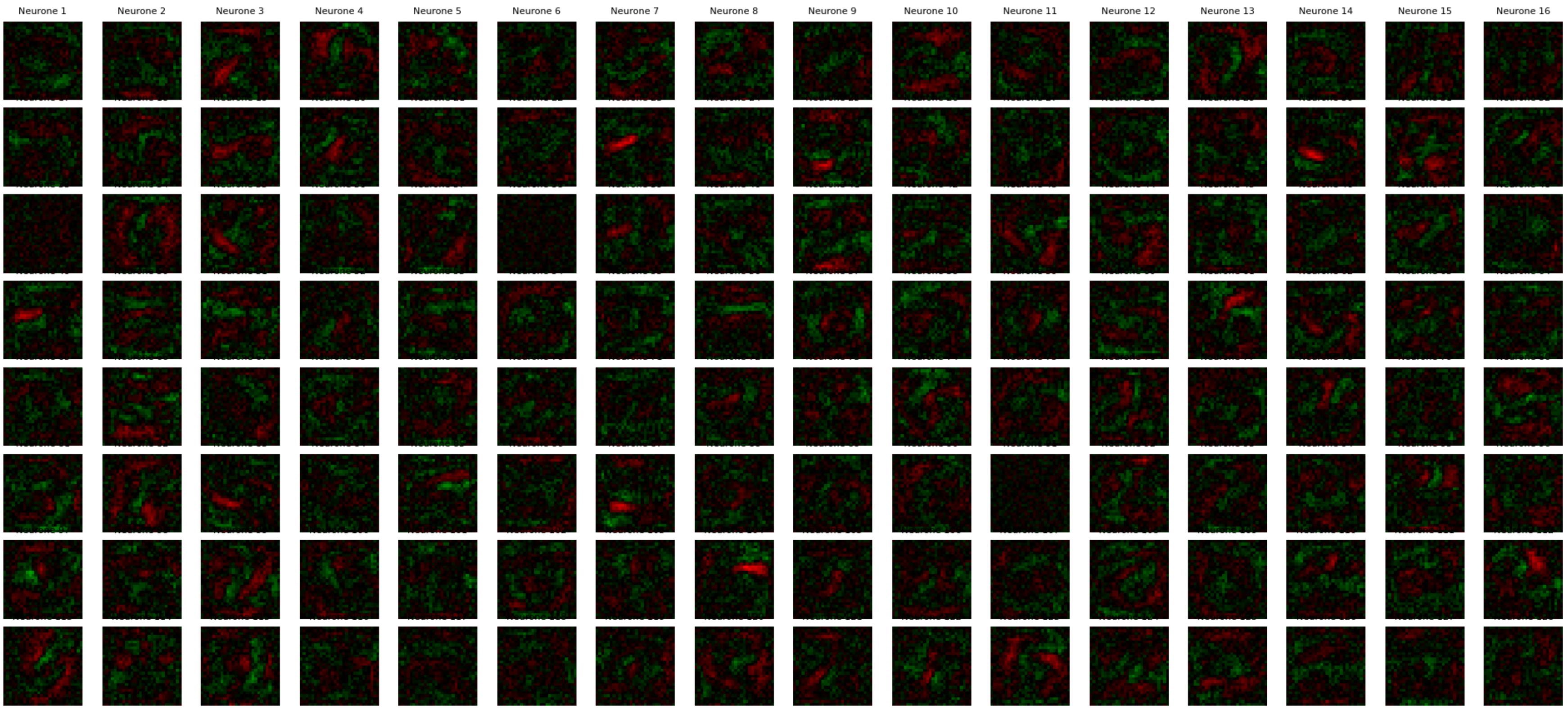
- Pesi Positivi (**Verde**): Rafforza l'attivazione di quel neurone
- Pesi Negativi (**Rosso**): Inibisce l'attivazione di quel neurone.
- Pesi Neutri (Nero): Non ha influenza significativa sul neurone.



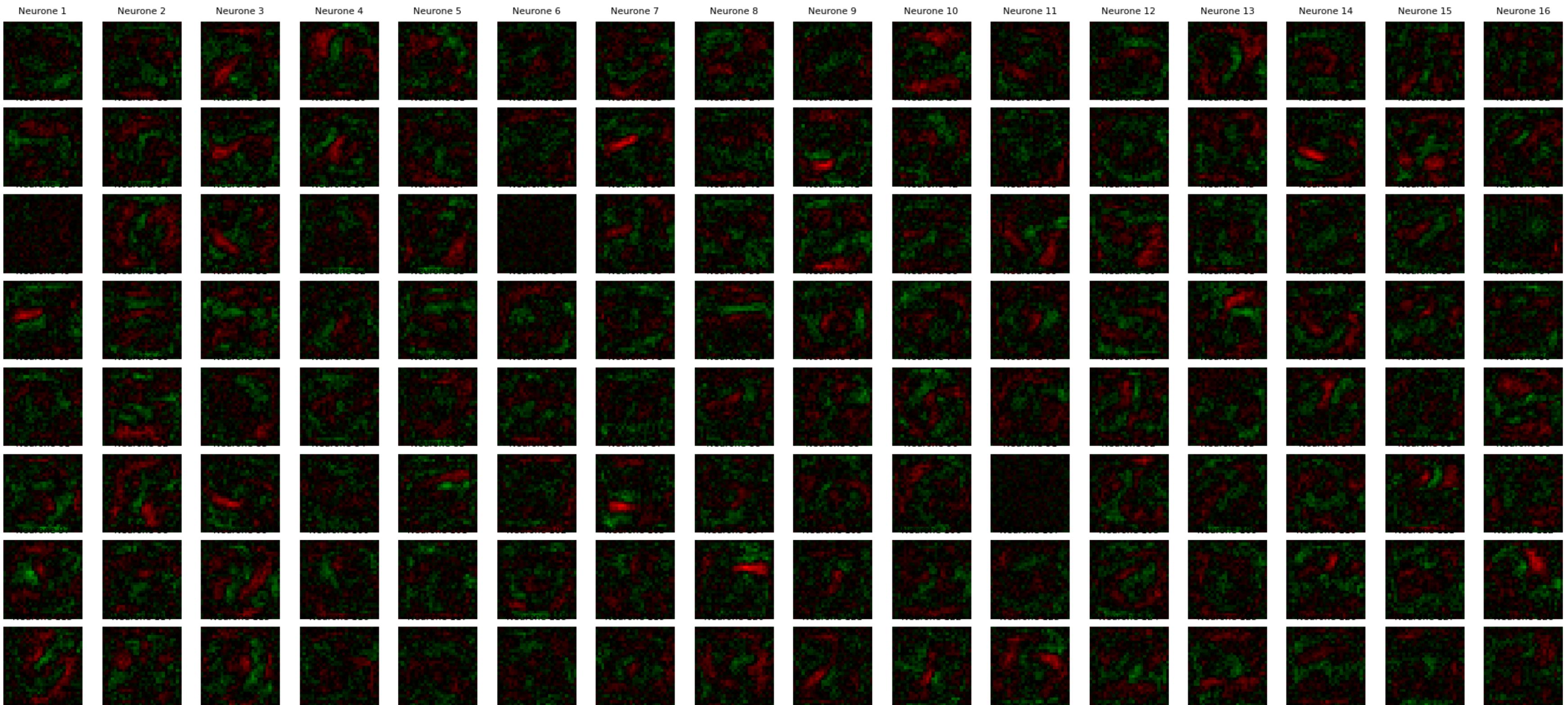
Evoluzione Pesi del Neurone (Epoca 1)



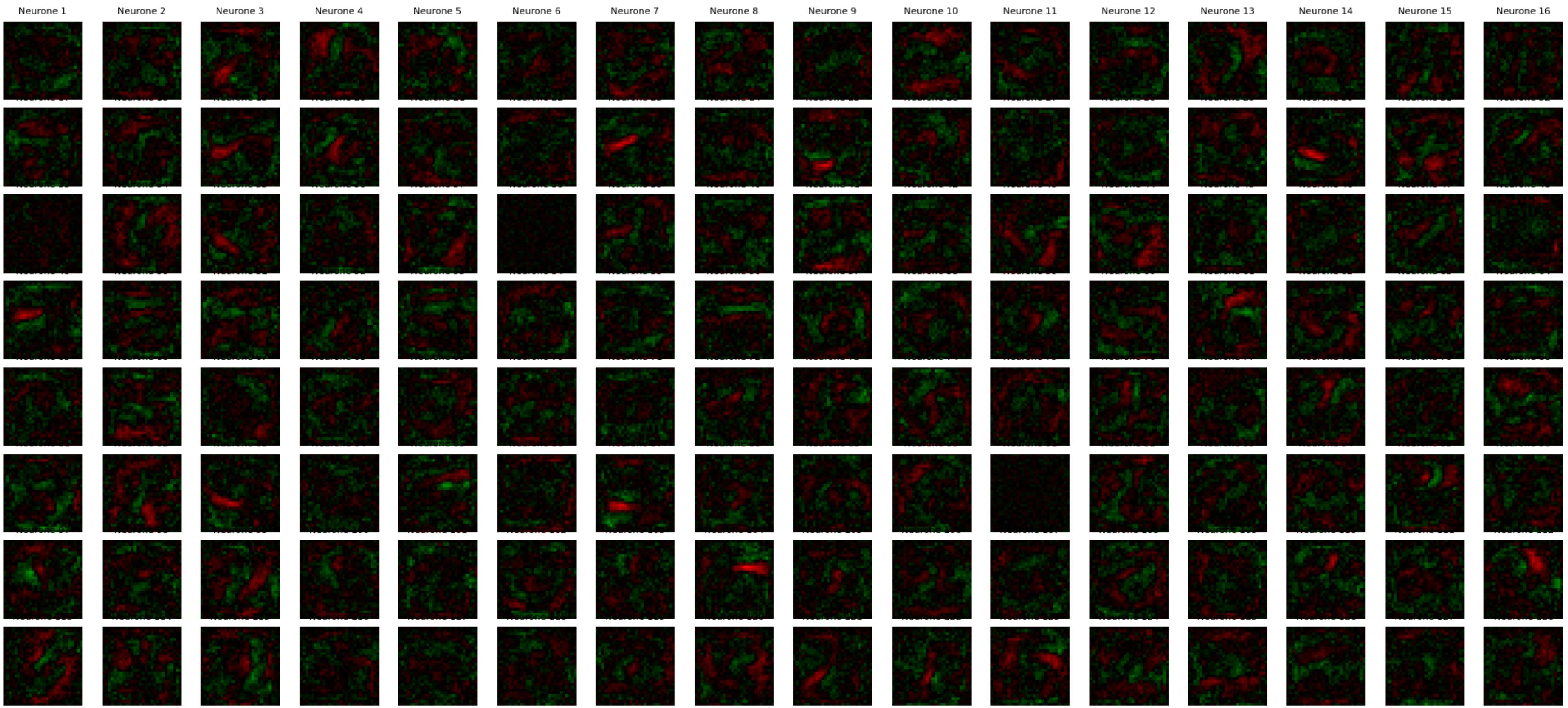
Evoluzione Pesi del Neurone (Epoca 2)



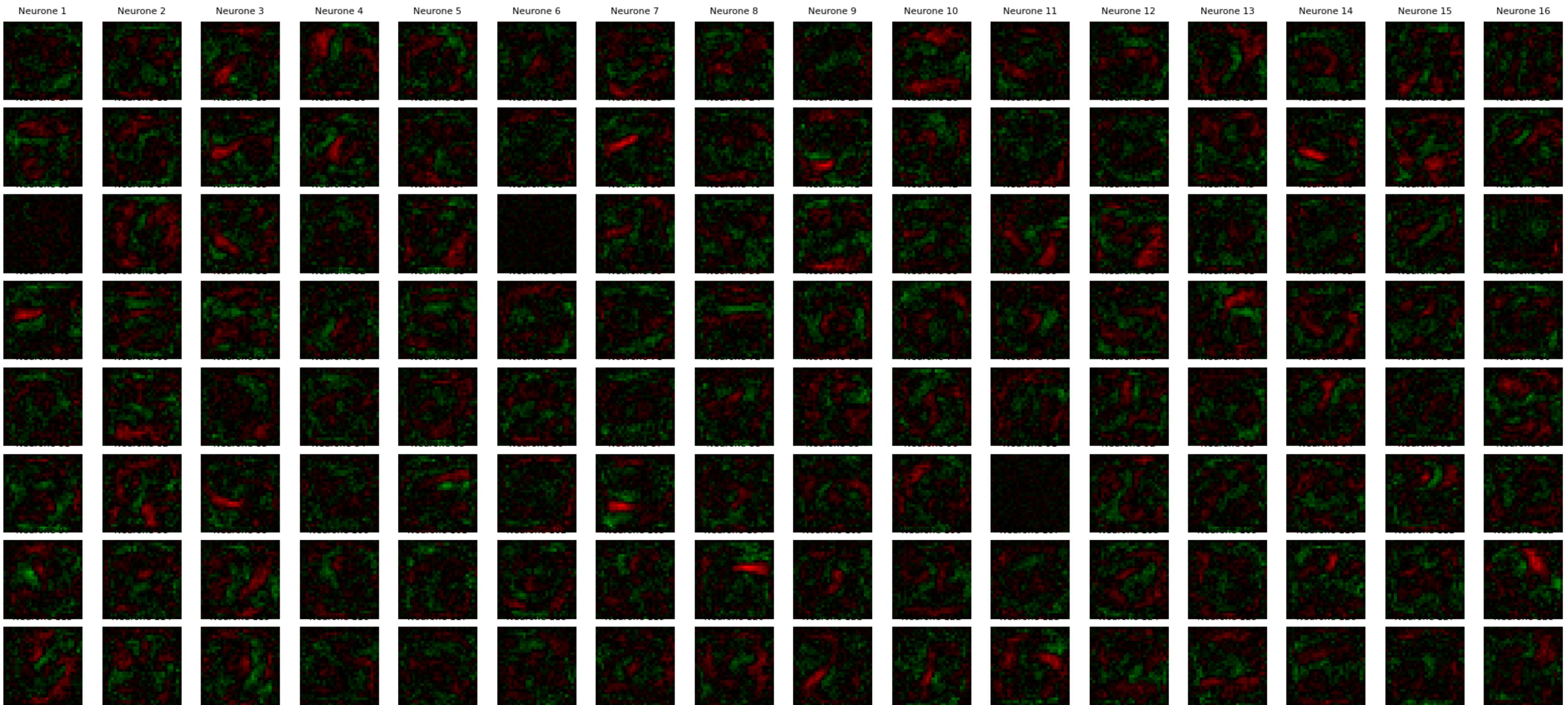
Evoluzione Pesi del Neurone (Epoca 3)



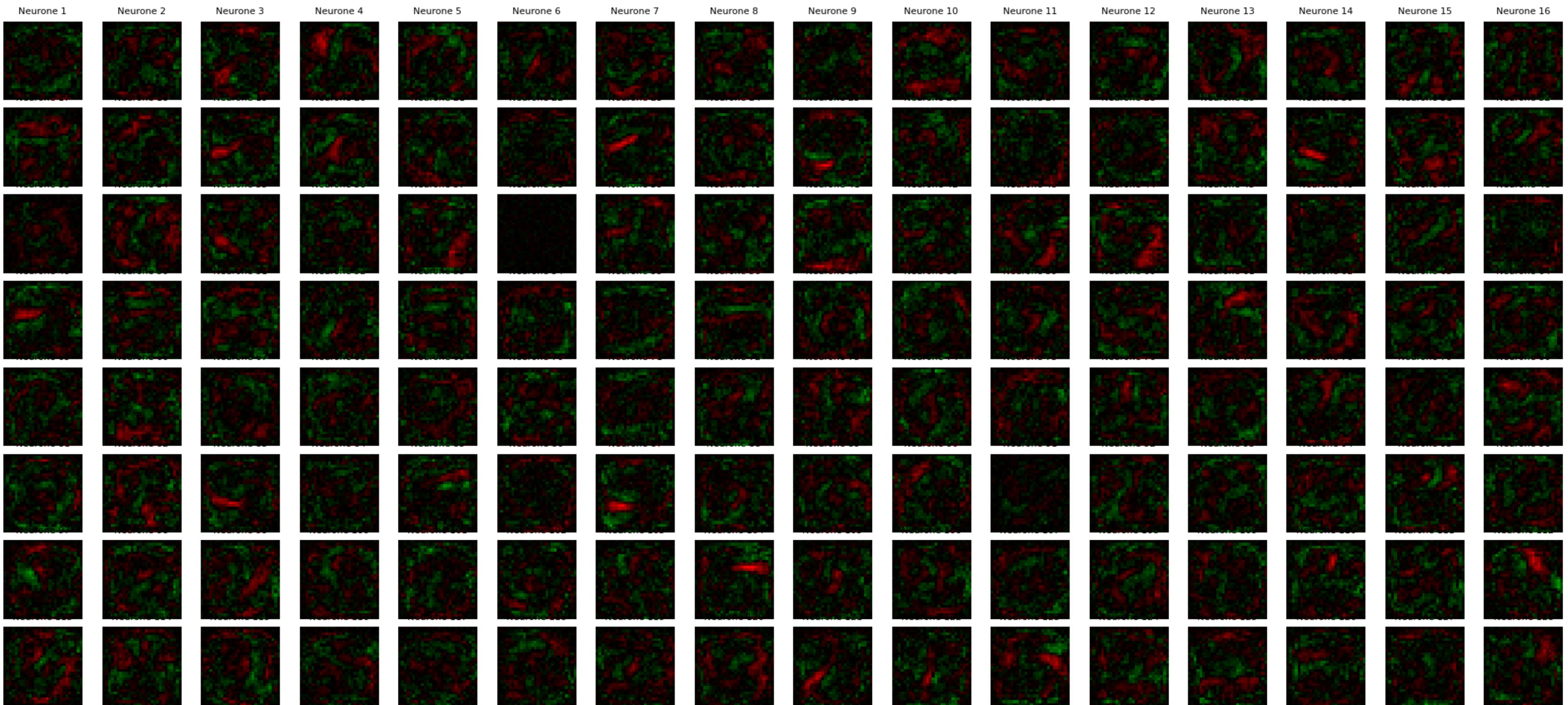
Evoluzione Pesi del Neurone (Epoca 4)



Evoluzione Pesi del Neurone (Epoca 5)



Evoluzione Pesi del Neurone (Epoca 10)



Conclusione

La rete neurale ha commesso soltanto
230 errori su 10.000 immagini del
dataset.

L'MLP ha quindi raggiunto
un'accuratezza finale del 98%,
confermando l'**efficacia del modello**.

Sitografia

- [Introduzione alle reti neurali feed-forward](#)
- [Rete neurale feed-forward](#)
- [Convolutional Neural Networks](#)
- [Recurrent Neural Networks](#)
- [LSTM Explained](#)
- [Transformer Neural Networks](#)
- [GANs](#)
- [Graph Neural Networks](#)
- [Neural Networks and Deep Learning](#)
- [Neural Networks](#)
- [Backpropagation](#)
- [Optimizing Gradient Descent](#)
- [Gradient Descent](#)
- [Deep learning](#)
- [Activation Functions](#)
- [Adam optimizer](#)
- [Deep learning optimizer](#)





**Grazie per
l'Attenzione**